Introduction to Python Programming

2005 – 2007

Contents

List of Figures

Chapter 1

Unix introduction

I'll begin these musings with the briefest of brief introductions to the operating system called Unix. If you already know all about it, or if you're using Windows, you can safely skip this chapter.

You execute unix commands through a *shell*. A shell is a program which lets the user input commands and then executes these commands. So, in order to start doing something, you need a shell. It could be an *xterm*. When you log into the DAIMI system, a shell probably starts automatically.

You can usually scroll through the commands you have issued so far by using the up- and down- cursor keys.

In the DAIMI system, your files and directories (folders) are organized in a tree structure. The *root* of this tree is your *home directory*: `/users/<your username>/`, and its branches are subdirectories. Inside this root directory and all its subdirectories, all your files reside. By convention, the tree is oriented upside down, so that "moving up" in the tree means moving one level closer to the root. Thinking in terms of ancestral trees, each directory in the tree has a unique *parent* directory (the one it resides in) but may have several *child* directories (the ones it contains).

Your shell opererates with a current *working directory* (WD) which is the directory where you currently "are". When you begin, the WD is your root directory. Any file or directory has a *path* which uniquely identifies it in the tree, e.g., **/users/chili/public_html/PBI05/unix.html**. A path can be absolute, like the one above, or it can be relative. If it's relative, the WD is taken as a starting point, so the path `Images/me_in_Nepal.jpg` refers to the file `me_in_Nepal.jpg` inside the directory `Images` inside the current working directory.

All files and directories have *permissions* associated with them: Permission to write, read, or ecxecute. You, as owner, can set these permissions. You can set different permissions for yourself and all others.

The first of the following two tables present some commands which I strongly suggest you learn to master immediately. The second presents some very useful commands that you'll probably eventually learn anyway.

In every filename, you may use a $\star$ as a joker to mean "any combination of characters". Thus, **ls *.py** lists all files in the WD ending with **.py**.

pwd	Gives the absolute path to the current WD.
cd <**dir**>	Changes the WD to <dir>.
cd ..	Changes the WD to the parent directory of the current WD (move up one level).
mkdir <**dir**>	Makes a new directory <dir> inside the WD.
rmdir <**dir**>	Removes the (empty) directory <dir>.
more <**path**>	Prints the contents of the file <dir> - press Space or the cursor keys to scroll. If you press **/** you can type a string, and when you press Enter, the (rest of the) file is searched for this string.
ls	Lists the contents of the WD.
ls <**dir**>	Lists the contents of <dir>.
ls -l	Like **ls** but lists the directory contents in tabular form with file information.
mv <**path1**> <**path2**>	Moves the file/directory <path1> : If the directory <path2> exists, <path1> is moved there. If not, <path1> is *renamed* to <path2>.
cp <**path1**> <**path2**>	Copies the file <path1> : If the directory <path2> exists, a copy of <path1> is created there. Otherwise, a copy of <path1> is created with the name <path2>.
rm <**path**>	Removes the file <path>.
∼**/**	Shorthand for your root directory.
∼<**username**>**/**	Same as the root directory of user <username>.
chmod <**mode**> <**path**>	Changes the permissions for the file/directory <path> according to <mode> which should be a sequence of letter/code/action(s) triplets: • The letter **a** means all users, the letter **u** means user (you only) • The code **+** means permission is granted, the code **-** means permission is denied • The actions **r**, **w**, **x** mean read, write, execute, respectively Thus, the mode **a+rx** means that all users are granted read and execute access.

<**cmd1**> | <**cmd2**>	*Pipes* the output from <cmd1> into <cmd2>. I.e., the output from the first program is sent as input to the second program.
<**cmd**> > <**path**>	*Redirects* the output from <cmd> into the file <path> - since it is redirected to the file rather than to your screen, you won't see any output. If <path> already exists, it will be overwritten!
cat <**path**>	Prints the entire contents of the file <path>.
man <**cmd**>	Show documentation for the command <cmd>.
<**cmd**> | **tee** <**path**>	Saves the output from <cmd> in the file <path> (which is overwritten if it exists), but also show the output on your screen.
top	Lists the programs running on your computer currently demanding the most CPU power. Each program (job) name is in the right-most column, its owner is in the second column. If your computer is running very slowly because someone else is running a heavy job on it, you can send an email asking him/her to stop it. Type 'q' to quit `top`.
grep <**pattern**> <**path**>	Report all lines in the file <path>, or in all files in the directory <path>, containing the string <pattern> (put quotes around it if it contains special characters).
grep -c <**pattern**> <**path**>	As above but only report the *number* of lines containing the pattern string.
grep -f <**patternfile**> <**path**>	Search in the file <path>, or in all files in the directory <path>, for all patterns found in the file <pattern>. Each line is interpreted as a pattern. `grep` is a *very* useful command; do a `man grep` to learn more.
<**cmd**> | **cut -d "**<**char**>**" -f**<**list**>	Split each line in the output of the command <cmd> into fields delimited at each occurrence of the character <char> and report only those fields listed in <list>. E.g., **cat mylist.txt** | **cut -d " " -f2,4** lists the second and fourth fields of each line in the file `mylist.txt` after splitting it at each space.
<**cmd**> | **sort**	Sort the output from <cmd> line by line.
<**cmd**> | **sort -u**	Sort output but ignore duplicate lines.

You can guess the remainder of a file or directory name you have started typing with the Tab key: Type the first few letters of the name and the press Tab; then your shell auto-completes the name. If there are several options, it will beep and expect more characters to be typed in.

You can use `.` for the current working directory. This is nice, e.g. when you wish to copy or move something "here".

If you want to learn more or need further explanation, an excellent series of Unix tutorials for beginners is available at `http://www.ee.surrey.ac.uk/Teaching/Unix/index.html` The first five tutorials cover the most basic stuff.

Chapter 2

Introduction via the interactive interpreter

There are two ways to execute computer programs: via an interpreter, and via a compiler. In compiled languages like C and Java, you first have to translate your entire program into something which you can then execute directly. This 'something' is in fact your program translated into a language which your computer understands[1]. Compiling takes some time, but the following execution is pretty quick.

Interpreted languages (like Python) don't work that way; when you run your program, the Python interpreter reads one command at a time, then translates it into something the computer understands, then executes it, then reads the next command. The translation and execution are interlaced, and therefore it normally takes a little longer to run. On the other hand, you don't need to wait while your program is being compiled.

In Python, there are two ways to run the interpreter. You can give it a complete program, or you can supply it with commands one by one, seeing the results of each command as you go along. We'll start by talking about this interactive interpreter.

2.1 The interactive interpreter

On Unix you simply type `python` in your shell to start the . You'll get something like this:

```
Python 2.3.2 (\#1, Nov 13 2003, 20:02:35)
[GCC 3.3.1] on linux2
Type "help", "copyright", "credits" or "license" for
more information.
>>>
```

[1].. or, rather, which is pretty close to something your computer understands.

The interpreter is now at your command, waiting for something to do. If you type something not Python, it will complain:

```
>>> chocolate
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'chocolate' is not defined
```

The last line makes sense; don't worry about the rest, we'll get to that.

For starters, we can use the interactive interpreter as a calculator. Try

```
>>> 4*2+0.5-(10-2)
0.5
```

Notice that `4*2` is calculated first and then the result is added to `0.5` before the parenthesis is calculated and the result subtracted. In other words, *the multiplication operator takes over the addition operator* if you don't put in any parentheses: `4*2+0.5` yields 8.5, not 10, because it is calculated as `(4*2)+0.5`.

All operators have their place in this hierarchy of precedence[2]; I suggest you simply add as many parentheses you need to make your expression clearly readable and ambiguous to you. It will be seconds well spent.

A few more examples:

```
>>> 2**8
256
>>> 10%3
1
```

Perhaps you are unfamiliar with the `**` operator. It is the : `2**8` means 2^8.

The `%` operator is Python for the : `10%3` calculates the remainder of 10 divided by 3. Mathematically, for `x%y` you'll get the number $x - \lfloor x/y \rfloor * y$. Recall that $\lfloor x/y \rfloor$ (floor division) is the highest integer less than or equal to x/y, so $\lfloor 10/-3 \rfloor = -4$. That means you can use modulo for negative numbers also:

```
>>> 10%-3
-2
>>> -10%3
2
```

[2]If you want to know the details, go to `http://www.python.org` and search for "operator precedence", then click on the first link which appears.

```
>>> -10%-3
-1
```

Finally, a look at the division operator `/`. If you divide two integers, it performs floor division and rounds down (discards any fractional part of) the result to an integer (if it isn't already an integer):

```
>>> 20/5
4
>>> 20/6
3
>>> 6/7
0
```

If at least one of the arguments is a floating point number, so will the result be; i.e., `/` then performs "true" division:

```
>>> 1.5/3
0.5
>>> 20.0/5
4.0
```

It may seem stupid to you that the division operator is ambiguous. The Python designers thought so too, and from version 2.0, they introduced an extra division operator, `//`. The intention is that `/` performs true division (i.e. no silent rounding), and `//` performs floor division. If you want this functionality, you must tell the Python interpreter explicitly with a magic spell:

```
>>> from __future__ import division
>>> 6/7
0.8571428571428571
>>> 6//7
0
```

If you don't actively do anything do import the `//` operator, `/` will behave as explained first and perform floor division.

2.2 A first look at functions

A priori, Python only knows the basic calculus operators, but if you tell it you need more advanced stuff, you'll get it. Using another magic spell (to be explained in Chapter 5 — but note the similarity of the two magic spells you've seen), we gain access to a bunch of mathematical functions and constants:

```
>>> pi
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'pi' is not defined
>>> from math import *
>>> pi
3.1415926535897931
>>> e
2.7182818284590451
>>> exp(3)
20.085536923187668
>>> log(10)
2.3025850929940459
>>> log10(10)
1.0
```

The magical `import` line tells Python that we want to import everything (denoted by the $\star$) from a code module with mathematical content. Having done that, we suddenly have access to the constants π and e. Further, we can use the exponential function `exp`, the natural logarithm `log` as well as the base 10-logarithm `log10`, and others.

Computing the area of a circle is as easy as π; in fact, let's compute the areas of three circles with radii 3, 5 and 7 using the formula $a = \pi r^2$:

```
>>> pi*3*3
28.274333882308138
>>> pi*5*5
78.539816339744831
>>> pi*7*7
153.93804002589985
```

This is unelegant. We don't want to type almost identical, utterly tedious calculations over and over again. The solution is to write our own *function* which computes the area of a circle given its radius r. Here's how (now with line numbers on the left – they won't appear on your screen):

```
1  >>> def circle_area(r): return pi*r*r
2  ...
3  >>> circle_area(3)
4  28.274333882308138
5  >>> circle_area(5)
6  78.539816339744831
```

```
7 >>> circle_area(7)
8 153.93804002589985
```

Try defining this function in your interactive interpreter. (After typing in the first line, you'll see the three dots. Just type return, and you're back with the >>> marker. And don't terminate the session just yet, you'll need the function again in a moment.)

In line 1, we define a function called `circle_area`; `def` is short for "define". A function name may contain letters and numbers and underscore characters (_) and cannot begin with a number.

After the function name, the function's *parameters(s)* appear in parentheses. The parameters will hold whatever you want to "send to" the function for it to deal with in some way. To calculate the area of a circle, you need to know the radius of it, and so our function `circle_area` needs one parameter which we call `r`. Then follows a colon, and after that comes the *body* of the function, i.e. the code which performs the job. The way to return the result is to simply use the keyword `return` followed by whatever needs to be returned; in this case π times the given radius r squared.

Then, on lines 3, 5 and 7, we *call* the function with arguments 3, 5, and 7, respectively, and Python prints the results. A function is called by typing its name followed by the desired argument(s) in parentheses.

Say we want to be able to calculate the side length of a square with the same area as a circle with a given radius r. That is, we want s such that $s^2 = \pi r^2$, or $s = \sqrt{\pi r^2}$. We define a new function `q` for this purpose, and rather than calculate the circle's area again, we can simply use the function we already have:

The famous mathematical problem called the *Quadrature of the circle* was defined by G(r)eeks more than 2000 years ago: Is it possible using only a ruler and a pair of compasses to construct a square with the same area as a given circle? The answer is no because you'd have to draw a line of length $r\sqrt{\pi}$, an irrational number.

```
>>> def q(r): return sqrt(circle_area(r))
...
>>> q(10)
17.724538509055161
>>> q(5)
8.8622692545275807
```

Here we use the `math` function `sqrt` for computing the squareroot of something. So in fact, our new function `q` calls two other functions: `sqrt` and our own newly defined `circle_area`. In this case, first the expression `circle_area(r)` is evaluated, i.e. the function `circle_area` is called with argument `r`, and the result in turn is given to the `sqrt` function which then computes the final result.

2.3 Introduction to variables

Look again at the way one defines a function:

```
def circle_area(r):  return pi*r*r
```

In the parentheses, the name of the function's parameter is given. In the body of the function, the *argument* with which the function is called is referred to by this name. When, e.g., the function is called with `circle_area(5)`, the argument value 5 is bound to the parameter name `r` inside the function body, and so every reference to `r` yields the value 5. Since the value represented by the name `r` in the function body can vary depending on the argument of the particular function call, `r` is also called a *variable*.

Variables referring to parameters in function definitions are *local variables*: They can only be used inside the function body. Try:

```
>>> def double_up(x): return 2*x
...
>>> double_up(5)
10
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'x' is not defined
```

Again we get the `NameError` (plus the secretive stuff). Outside the body of `double_up`, we can't refer to `x`. And if you think of it, it makes perfect sense: `x` represents whatever value the particular function `double_up` is called with. It's meaningless to refer to `x` when not calling its mother function. I'll define local variables more formally later.

Of course you can create your own variables as you go. The value of *Avogadro's constant* is $6.02214199 \times 10^{23}$. It is defined as the number of atoms of the same type needed to obtain a weight in grams equal to the atom's atomic weight. For example, the atomic weight of oxygene is 32.0, and so if you weigh off 32.0 grams of oxygene molecules, you'll have $6.02214199 \times 10^{2}3$ of them. Rather than typing this wild value each time you need it, you could define a new variable holding the value:

```
>>> avogadro = 6.02214199e23
```

This is also called *assignment* of a value to a variable (in Python, you're allowed to use the `e`-notation as a shorthand for "times 10 to the power of": `6.02214199e23` means $6.02214199 \times 10^{23}$). Now you've defined a new variable, and every time you use the name `avogadro`, you'll get the value it points to:

```
>>> avogadro
6.02214199e+23
>>> avogadro * 2
1.204428398e+24
>>> avogadro+avogadro
1.204428398e+24
```

You can alter this value later (either on purpose or in error); simply assign a new value to the same name. You can also introduce a variable without actually assigning it a value. That is done by assigning it the dummy value `None`.

Using the up/down arrow keys, you can surf the history of commands issued to the interpreter up until now. That's particularly useful when you want to re-execute some long command or a slightly modified version of some command.

The same rules apply to variable names as to function names: Letters, numbers, and underscores; first character can't be a number. In fact, these rules apply to all Python names in general: Python doesn't care what a name actually points to/identifies, whether a function, a number, or something else. A name is a name, and those are the naming rules. Ordnung muß sein.

When you assign some value to a variable, a reference is made to the chunk of memory representing this value from the variable name (which is also some chunk of memory). It's important to make the distinction between variable name and value: They are not the same thing, even though the assignment symbol = might signal just that; the assignment creates an association, a *reference*, between the variable name and its value. Thus, if you assign a new value to the variable, the old value may live on in memory. And, more subtly, several variables may be assigned *the same value* — not just copies of the value, but the exact same, physical chunk of memory representing the value. Don't worry too much about this for now; I'll make this point again in Chapter **??**.

2.4 Functions `dir` and `help`

Now as we've imported a lot of math functions, defined a few functions of our own, and created a variable or two, perhaps it would be nice to get an overview of what we know. Or rather, what Python now knows. Such a list is available through the function `dir`:

```
>>> dir()
['__builtins__', '__doc__', '__name__', 'acos', 'asin',
'atan', 'atan2', 'avogadro', 'ceil', 'circle_area',
'cos', 'cosh', 'degrees', 'double_up', 'e', 'exp',
'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp',
```

```
'log', 'log10', 'modf', 'pi', 'pow', 'q', 'radians',
'sin', 'sinh', 'sqrt', 'tan', 'tanh']
```

`dir` is short for "directory", and calling this function gives you a directory of identifiers (names identifying something) in quotes that Python knows at the time of the call. They are:

- Identifiers which Python is "born with".
- Imported identifiers.
- Identifiers you have created.

The inherent ones are `__builtins__`, `__doc__`, and `__name__`. We created `circle_area`, `q`, `double_up`, and `avogadro`, and all the rest are those imported from the `math` module: `pi`, `e`, `log`, `sin`, `hypot`, etc.

You, being all bright and pedantic, have of course noticed that the function `dir` itself does not appear in this list. Well right you are, and good that you noticed. The weird name `__builtins__` is in fact a module just like `math` holding yet another long list of identifiers. When you use some name, Python eventually looks in this module to find it. Try:

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError',
'DeprecationWarning', 'EOFError', 'Ellipsis', [...],
'abs', 'apply', 'basestring', 'bool', 'buffer',
'callable', 'chr', 'classmethod', 'cmp', 'coerce',
'compile', 'complex', 'copyright', 'credits',
'delattr', 'dict', 'dir', 'divmod', 'enumerate',
'eval', 'execfile', 'exit', 'file', 'filter', 'float',
'getattr', 'globals', 'hasattr', 'hash', 'help', [..]
'vars', 'xrange', 'zip']
```

(If you actually *do* try that, your list will be even longer; I've truncated it some). And tadaa indeed, here you spot `dir`. As I showed you, `dir()` will give you a global list of known identifiers while `dir(<module>)` gives you the identifiers defined in `<module>`.

Check the list for another built-in function called `help`. Calling it with no arguments starts an online help utility (which you exit by typing "q"), but you can also call it with some identifier as argument, and if documentation on the identifier is available, it will be printed. You may have wondered about the `math` function `hypot`. Go ahead and ask Python:

```
>>> help(hypot)
```

```
Help on built-in function hypot:

hypot(...)
    hypot(x,y)

    Return the Euclidean distance, sqrt(x*x + y*y).
```

(Press space and you're back with the interpreter). This info tells you that `hypot` takes *two* parameters x and y, and that it returns $\sqrt{x^2 + y^2}$; i.e. the length of the hypothenuse in a right-angled triangle with cathete lengths x and y. Hence the name `hypot`. In due time, you'll see how to attach such documentation info to your own functions.

Now quit the interactive session. (If you're running Unix, you quit the interactive interpreter by pressing `Control` + `d`. Using `Control` + `z` suspends it, leaving it running "in the background", slowing your computer down). If you start a new session immediately, you start from scratch. Every identifier you created or imported is gone. To avoid that, you put your Python code in a file. Every time you want your code executed, you simply hand the file to the Python interpreter rather than start it in the interactive mode. Say you create a file `firstprogram.py` in your favorite text editor with only one line in it: `def f(x):  return x+7`. To have this code executed, i.e. to create a running Python program which defines the function `f`, type this in your shell:

```
python firstprogram.py
```

Nothing happens, but no news is good news here. Python reads the file and executes the code one line at a time, just as if you had typed them manually in an interactive session. This program defines the function `f`, then proceeds to do nothing else, then quits. However, if you add the line `f(3)` to your program and run it again, still nothing happens. Had you done that in an interactive session, the result 10 would have been printed. The reason is that when running interactively, the Python interpreter expects that any you want to see the result it calculates. The result is 10, so 10 is printed. When running a program, the result is not immediately printed — you have to ask it to since it is not obvious that printing it is always the desired way to return the result. In the following chapter, you'll see how.

One more thing about the interactive interpreter: It's a great help while you're writing programs. It's an easy and immediate way to test some Python statement, e.g. in case you're unsure of the exact syntax. Another nice feature is that you can tell Python to go into interactive mode when done running a program by giving it option `-i`, like so:

```
python -i taxreturn.py
```

The great thing about that is that all names from the program are transferred to the interactive session: You inherit every identifier "alive" at the termination point of the program (if the program terminates, that is). Thus, you can inspect variables and check that they have the value you expect, etc. Makes debugging a little less frustrating.

Now you know how to:

- Execute commands as you type them in the interactive Python interpreter
- Do mathematical calculations
- Import common math functions and write your own
- Call functions and have them call each other
- Use and create variables
- Call `dir` and `help`

Chapter 3

Strings

Python is designed to work with characters as its primary form of data. It provides a comprehensive set of basic easy-to-use functions for manipulating characters and text, and you've also got more advanced functionality, e.g. for handling regular expressions (very nifty stuff, you'll like it). Before we get to all that, let's go over the basics.

3.1 Strings and how to print them

A *string* is a sequence of characters. A string is surrounded by quotes: You can use `'` and `"`, and if your string extends over several lines, you use triplets of either one. Here's a small program to demonstrate:

```
1  # This is a comment.
2  s1 = "Play it, Sam."
3  s2 = 'See you next Wednesday.'
4  s3 = '''All work and no play makes Jack a dull boy. All w
5  ork and no play makes Jack a dull boy. All work and no pl
6  qy makes Jack a dull boy.'''
7
7  print s3
8  print
9  print s1, s2
10 print s1 + s2
11 print s1,
12 print "You can't handle\nthe truth!"
```

Figure 3.1: Program `strings.py`.

Here I define three variables `s1`, `s2`, and `s3`. Up until now, you've only seen variables pointing to ("containing") numbers, but variables can point to anything and here they point to strings.

Look at lines 7ff. The command `print`, not surprisingly, prints a string. `print` always concludes with a newline (unless you put a comma at the end,

see below), so not giving it a string to print results in an empty line. Printing a variable means printing the *value* of that variable, not its name. Of course you can also print a string directly as shown in line 12; if a string itself contains one kind of quotes, use the other kind as delimiters. Line 12 also contains a `\n`: This so-called *escape sequence* stands for a newline (the `n` is "escaped" by the `\` to mean something else). See some other escape sequences in Table 3.1. `print` is a *keyword* in Python: That means that this name is reserved and that you're not allowed to use it for a variable. In the example programs of this text, all keywords will be written in blue.

`\n`	New line
`\\`	Backslash
`\'`	Single quote
`\"`	Double quote
`\t`	Tab

Table 2.1: Escape sequences.

Printing two strings separated by a comma prints the concatenation of the two strings with a space in between. If you don't want the space, another way is to concatenate the strings using `+`. (What you witness here is called *operator overloading*: The addition operator `+` is used both for adding two numbers and for concatenating two strings. That's neat! Since `+`, the symbol for number addition, is also an obvious, intuitively understandable symbol for the string concatenation operation, it's nice that's it's not a problem that the symbol is already "taken". There's a caveat, though, which you'll experience in the next section). If you put a comma after the string you want printed (as in line 11), you'll get a space *instead of* the newline which `print` otherwise makes.

Also observe the comment in line 1. A comment begins with a `#`, and all comments are ignored by Python (as are empty lines): When Python encounters a `#`, it skips the rest of the line. I strongly encourage you to put loads of comments in your programs. Use them for documentation; if it's not clear what's going on, put a brief explanation in a comment. Comments will be a big help when you look at the program again in a week, and in six months, without them it will be annoyingly time-consuming for you to figure out what the program does. And impossible for anyone else. In fact, you can hardly write too many comments (save "`don't forget mom's birthday`" and "`this doesnt work damn damn python sucks`" and the like).

Here's how the output looks if you run the program (note the result of the comma in the program's line 11):

```
All work and no play makes Jack a dull boy. All work
and no play makes Jack a dull boy. All work and no
plqy makes Jack a dull boy.

Play it, Sam. See you next Wednesday.
Play it, Sam.See you next Wednesday.
Play it, Sam. You can't handle
the truth!
```

Figure 3.2: Output from program `strings.py`.

3.2 Indexing strings

Since a string is a sequence of characters, it's possible to manipulate *parts* of it through subscripting or *indexing*. The characters of the string are numbered from 0 and up to the length of the string minus 1, and so you can ask for the character with index 0 to get the first character, the substring consisting of the characters with indices 4 to 9, etc. You index a string using square brackets `[]`. The notation is demonstrated in this example:

```
>>> s = "How about .. Russian?"
>>> print s[0]
H
>>> print s[4:9]
about
>>> print s[13:]
Russian?
>>> print s[:3]
How
>>> len(s)
21
```

There are several important things to note here, so I'll make a list:

- Indices start with 0.

  I'm sorry, they just do. First character has index 0, second character has index 1, etc. There are good reasons. There must be, right?

- You can *slice* out a part of a string `s` with `s[from:to]`.

  The rule is: "from first index to *but not including* last index", so you'll need to add 1 to the index of the last character you want to include in your slice. This makes sense, though: the length of the substring you get with `s[a:b]` will be $b - a$.

- You can leave out the first or last index of a slice (or both).

  `s[a:]` is the substring of `s` ranging from index `a` to the end
  `s[:b]` is the substring of `s` ranging from the start and ending with index `b-1`.

- To find out the length of a string, you have the built-in function `len`.

  Thus, the last index in any string `s` is `len(s)-1`.

- With slices, you always get a *new string*; the original string is untouched.

It's tedious to have to write `s[len(s)-1]` to get to the last character in a string `s` — it feels coarse to an aesthetic like you. Luckily, Python offers a clever remedy. In fact, the characters of a any string `s` are numbered both ways: Left to right from 0 to `len(s)-1`, but also right to left from -1 to `-len(s)`! That's an extremely pleasant feature allowing, e.g., easy and elegant access to the last 7 characters in some string. Observe:

```
>>> t = "The Beatles"
>>> print t[-1]
s
>>> print t[1:-1]
he Beatle
>>> print t[-7:]
Beatles
```

Using slicing, you can carve out any chunk of a string. Using *extended slicing*, you can do a sort of intelligently picky slicing. Extended slicing is just like slicing but with a third argument following an extra `:`. This third argument indicates the *stepping order* of characters to be chosen from the slice given by the first two arguments. A stepping order of *n* means "go through the slice in steps of *n* starting at the character given by the first argument, joining each character you hit in a new string". Lookie here:

```
>>> s = "Alabama and Alaska"
>>> s[:7]
'Alabama'
>>> s[:7:2]
'Aaaa'
>>> s[1:7]
'labama'
>>> s[1:7:2]
'lbm'
```

The first 7 characters of the string `s` are sliced out with `s[:7]` (leaving out the *from* index) yielding `Alabama`. Now to pick from this slice only every second character (yielding four `a`'s), use 2 as the third argument in the slicing operation: `s[:7:2]`. If you instead look at the slice `s[1:7]` and again only pick every second character (`s[1:7:2]`), you get `lbm`. Now watch this:

```
>>> s[::2]
'AaaaadAak'
>>> s[::-1]
'aksalA dna amabalA'
```

Now `s[:]` yields a copy of the whole of `s` since leaving out the *from* and *to* indices is the same as `s` from one end to the other. Thus, `s[::2]` gives you every second character in the whole string (`AaaaadAak`). And now pay attention, because the next trick is useful, neat, and the fastest way to solve a specific problem: If the stepping argument is negative, you step *backwards* through the slice. Thus, `s[a:b:-1]` gives you the characters in `s` going backwards from index `a` down to but not including index `b`. Leave out an index and you'll go all the way to or from the end. Thus, if you leave out both indices as in `s[::-1]`, you'll get every character in `s` starting at the last character and going backwards to the beginning — in other words, you'll get `s` *reversed*. `s[::-2]` would give you every second character in the reversed `s`. Prego!

Thus, ordinary slices are like extended slices with a stepping order of 1: `s[a:b]` is the same as `s[a:b:1]` for any string `s`. You always go **from** the first argument **to** but not including the second argument in steps given by the optional third argument (which defaults to 1). Here are a few more examples; as one of them shows, the resulting slice is empty if `b` is not smaller than `a` and the stepping order is negative — you can't go from index 2 "down to" index 4.

```
>>> s = "01234567"
>>> s[2:4:-1]
''
>>> s[4:2:-1]
'43'
>>> s[4::-1]
'43210'
```

3.3 Input from the user

One of Python's built-in functions is called `raw_input`. It is used to receive textual input from the user. When called, it prints an optional query string, waits for the user to type something concluding with the Enter key, and then returns a string containing whatever text the user entered. Start the interactive interpreter and try this:

```
>>> a = raw_input("Input first integer: ")
Input first integer: 7
>>> b = raw_input("Input second integer: ")
Input second integer: 3
>>> print a+b
73
```

The query string is printed (`Input first integer:`), and then you type something (7, also printed). Python will wait until you press Enter, then move

on to the next call to `raw_input` and print query string and input (3). Then it prints the sum.. 73? How's that for a shifty interpreter?! Well, in fact you have been fooled by the much-acclaimed operator overloading. We expect the user to enter two integers. But `raw_input` *always* returns a string — in this case the strings `"7"` and `"3"` rather than the numbers 7 and 3. And since the `+` operator also works for a pair of strings, Python doesn't complain and just concatenates them.

We need to convert the strings into integers so that `print a+b` will produce the expected result. Another built-in function does exactly that: `int` converts its argument into into the integer equivalent (of course, you'll get an error if you try `int`'ing something not representing a number). Thus, in the interactive interpreter use the cursor up-key to easily modify and reexecute the above code:

```
>>> a = int(raw_input("Input first integer: "))
Input first integer: 7
>>> b = int(raw_input("Input second integer: "))
Input second integer: 3
>>> print a+b
10
```

The call to `raw_input` blocks until the user has entered a string. Then `raw_input` returns, handing the string over to `int` which in turn converts it to an integer. Eventually, this time we get the expected result when we add `a` and `b`.

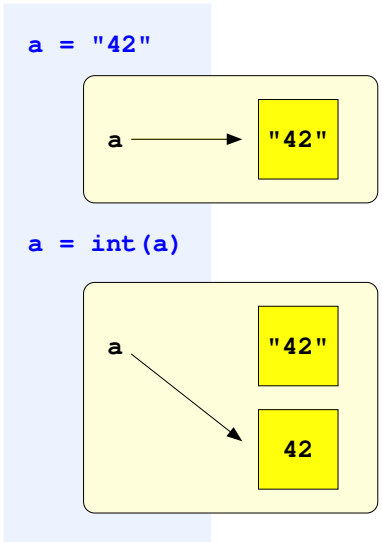


Figure 3.3: The function `int` creates a new integer object in its own memory location. The blue-shaded box is a program, the yellow-shaded boxes show the memory content after each Python statement.

Figure 3.3 shows how exactly `int` works through a different example. When Python executes the line `a = "42"`, it creates the new string `"42"`, allocates a new memory location for it, and creates a new variable name `a` pointing to this location. When executing `a = int(a)`, `int` converts the argument string into an integer, saves this integer in a new memory location, and returns a reference to this location which in turn is assigned to `a`. Thus, the old string `"42"` still exists in memory but is no longer accessible since no variables point to it. A program can't change an object's type or location; instead, a new object is created.

Python uses a *garbage collector* to dynamically get rid of unused objects lying around in memory. The garbage collector works while your program is running, recycling objects which are no longer reachable through any variables. Without it, you might run out of free memory, e.g. through the use of functions like `int`.

The idea behind `int` is that it should convert anything into an integer in a meaningful way if at all possible. Thus, strings like `"7"` and `"4182"` are straightforward, but in fact it also converts floating-point numbers, rounding down:

```
>>> int(3.79)
3
```

Don't push it though; its shrewdness has limits:

```
>>> int("3.79")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: invalid literal for int(): 3.79
```

It won't do two conversions for you, first from string to floating-point number and then from floating-point number to integer. To do that, you need the analogous built-in function `float`:

```
>>> float("3.79")
3.79
```

Representing floating-point numbers in a computer is not trivial, and accurately representing numbers with infinite fractional parts like 3.3333333... and π is obviously impossible (odds are your computer is finite like mine). Consequently, such numbers are in fact represented as *approximations*. Exactly how this problem is attacked is (way) beyond the scope of this text; it suffices to say that it's long-haired and has to do with computers' affinity for the binary number system. You'll occasionally feel the surface waves of the underlying battle in inexplicable oddities like these:

```
>>> float("3.78")
3.7799999999999998
>>> 10.0/3
3.3333333333333335
```

Refer to, e.g., `http://en.wikipedia.org/wiki/Floating_point` for more information about number representation.

3.4 Formatting the printed output

Figure 3.4 is a wee program which illustrates a few features you should know. Run it, and you'll get what is shown in Figure 3.5.

```
1 a = int(raw_input("Input a number between 0 and 10: "))
2 b = int(raw_input("Input a number between 0 and 10: "))
3 print "Histogram with 2 bars of width", a, "and", b, ":"
4 print "-"*a
5 print "-"*b
```

Figure 3.4: Program `printingvariables.py`.

```
1 Input a number between 0 and 10: 3
2 Input a number between 0 and 10: 8
3 Histogram with 2 bars of width 3 and 8 :
4 ---
5 --------
```

Figure 3.5: Output from program `printingvariables.py`.

First of all, note the neat way of producing a string consisting of a repetition of the same character (output lines 4 and 5). And second, note the not so neat space between the 8 and the colon in output line 3; it would look better without it: `Histogram with 2 bars of width 3 and 8:`. It's easy enough to explain: Putting a comma between two things to be `print`'ed inserts a space between them, as you know since recently. In this case (line 3 in Figure 3.4), we're printing a mixture of strings and integer variables separating them by commas, and that may often be the right way to do just that. But, there's no getting around that space which a comma inserts.

An alternative way to print a mix of variables and strings is to put the variables *inside* the strings. That's done by means of the *formatting operator* `%`. Here's a string formatting program illustrating its use:

```
1 a = int(raw_input("Input a number between 0 and 10: "))
2 print "Here are %d a's:"%a
3 print a * 'a'
4 b = 2*a
5 print "Here are %d b's:\n%s"%(b, b*'b')
```

Figure 3.6: Program `stringformatting.py`.

If a string is followed by the formatting operator `%` and an argument (or a list of arguments in parentheses), Python looks inside the string for special markers called *conversion specifiers*. The number of conversion specifiers should match the number of arguments given to the formatting operator. What happens is that Python replaces the first conversion specifier with the first formatting operator argument, the second with the second and so forth, scanning left to right. A conversion specifier consists of a `%` plus one or more characters specifying how exactly the argument should be inserted in the string and what kind of argument is expected. Confused? Look at the output (Figure 3.7) from the `stringformatting.py` program.

```
1 Input a number between 0 and 10: 4
2 Here are 4 a's:
3 aaaa
4 Here are 8 b's:
5 bbbbbbbb
```

Figure 3.7: Output from program `stringformatting.py`.

The program's line 2,

```
print "Here are %d a's:"%a
```

prints

```
Here are 4 a's:
```

The conversion specifier is `%d` which indicates that the corresponding formatting operator argument, `a`, must be a "signed integer decimal number" (an integer, simply). Line 5,

```
print "Here are %d b's:\n%s"%(b, b*'b')
```

sports two conversion specifiers, `%d` and `%s`, and two formatting arguments, `b` and `b*'b'`. It also contains a newline (`\n`). The first formatting argument is substituted for the first conversion specifier, and the second formatting argument is substituted for the second conversion specifier. Thus, the value 8 of the

integer variable `b` replaces `%d`, and you've already guessed that `%s` is for inserting another string in the mother string. Now `b*'b'` is not really a string, but it is an expression which *evaluates* to a string, and that's good enough. Therefore, as the raisin in the hotdog end, the string result `bbbbbbbb` of the expression `b*'b'` replaces `%s`, and the total result is this:

```
Here are 8 b's:
bbbbbbbb
```

As mentioned, the conversion specifier also controls *how* its argument is inserted in the string. Suppose you want the U.S. dollar equivalent of $53,75$ Danish Kroner; you'll want a result in dollars and cents, i.e. something with two digits after the comma, not a zillion as you'll get by default. Et voilà:

```
>>> print 53.75/6.27
8.57256778309
>>> print "%.2f"%(53.75/6.27)
8.57
```

The `%.2f` means "insert here a float and force two digits after the comma". The program in Figure 3.8 illustrates other examples of conversion specifiers; its output follows in Figure 3.9. The program also demonstrates that you can put several statements on the same line separated by comma. Only do this if the readability of your program doesn't suffer from it.

```
1 i = 65537; s = "Carpe diem"
2 print "Right justify int in field of width 9: [%9d]"%i
3 print "Left justify int in field of width 9:  [%-9d]"%i
4 print "Force 9 digits in integer:             [%.9d]"%i
5 print "Only 7 characters allowed in string: [%.7s]"%s
6 print "String in field of width 17: [%17s]"%s
```

Figure 3.8: Program `stringformatting2.py`.

```
Right justify int in field of width 9: [    65537]
Left justify int in field of width 9:  [65537    ]
Force 9 digits in integer:             [000065537]
Only 7 characters allowed in string: [Carpe d]
String in field of width 17: [       Carpe diem]
```

Figure 3.9: Output from program `stringformatting2.py`.

As you see, the numbers and the dot between the `%` and the letter code in a conversion specifier have different meanings depending on the type of the argument to be inserted. The best way to get the hang of this is through experimenting. Try playing around with the `stringformatting2.py` program: Test longer strings, different field widths etc. to see how exactly things work.

With formatting operators clogging your `print` statements, they'll tend to look hiddeous, but you'll learn to appreciate them. They are particularly useful for creating tabular output as you'll see later. You'll be using the conversion specifiers for strings and numbers almost exclusively, but there are several others; if interested, you might check `http://www.python.org/doc/2.4/lib/typesseq-strings.html`.

Oh, and by the way: If you need to include a percentage symbol in a string also containing conversion specifiers, use %%. Otherwise, Python gets confused thinking all %'s are conversion specifiers and that one of them can't be converted because the formatting operator lacks an argument:

```
>>> print "%d %"%10
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: incomplete format
>>> print "%d %%"%10
10 %
```

3.5 String methods

As explained, Python is designed for working with characters and strings. Therefore, the most common operations you can think of concerning strings have been made available as built-in functions. The way to call such a function on any string `s`, say the one called `capitalize`, is simply:

```
s.capitalize()
```

I.e., the formula is `<string>.<function>()`. Functions "belonging" to objects like this (as opposed to being functions in their own right) are called *methods* — you'll learn more about methods in a later chapter. Figures 3.10 and 3.11 are tables of the most important of the string methods; you might not completely grasp what all of them do, but when we get a little further ahead, you will. And you'll eventually use many of these methods oodles of times in the future.

A note on notation in the tables: In the method definitions, any parameters inside square brackets are optional. A definition is read left to right: When you meet a left square bracket, you either *ignore everything between it and its right partner*, or you just remove the brackets, keep what's between them, and read on. Thus a thing like `count(sub[, start[, end]])` means that the

method `count` always requires a `sub` parameter. After the `sub`, a left `[` follows whose right partner is at the very end of the parameter list (both colored red). Thus, either you ignore the rest of the parameter list to call `count` with only the `sub` parameter, or you skip the two bracket characters and read on. In that case, you get to the `start` parameter which you must then supply, and after that, you get to another pair of square brackets (colored blue) only holding an `end` parameter which you can then choose to ignore or keep.

In summary therefore, you can supply `count` with either a `sub` parameter, a `sub` and a `start` parameter, or a `sub`, a `start` and an `end` parameter. You can't supply an `end` without a `start`.

One of the most useful of all the string methods is `find`. Very often you'll be searching for some substring `s` inside a larger string `S`. Sometimes you'll want to find the first occurrence of `s` *after* some index *i*. Note the difference between these two statements:

```
>>> S = "abcdefg"; s="c"; i=2
>>> S.find(s, i)
2
>>> S[i:].find(s)
0
```

Both find the substring `c` but they report different indices. See why? `S[i:]` is a *copy* of the slice of `S` starting at index 2, and in this new string, the `c` occurs in index 0. It will almost always be the first version which does what you need — after all, the `c` *is* in index 2 in `S`, not index 0.

All these string methods can be called on any string `S` you create. Under some circumstances, you might need to call a string method using `S` as an *argument* to the method, rather than calling the method on, or through, `S`. This is possible with `str.<string method>`: E.g., `str.lower(S)` returns a lowercase version of `S`. I'll get back to this issue later.

Now you know:

- What a string is and how to index and slice it
- How to get input from the user with `raw_input`
- The functions `int` and `float`
- How to print strings and format the printed output
- About string methods

Method name and **example**	**Description**

`S.capitalize()`
 	
`>>> "exAmpLE".capitalize()`
`'Example'`
	Return a copy of `S` with its first character capitalized (and no others).

`S.center(width)`
 	
`>>> "abc".center(7))`
`' abc '`
	Return a string of length `width` in which `S` is centered, using spaces to "fill up" if necessary. If `S` is longer than `width`, return `S`.

`S.count(sub[, start[, end]])`
 	
`>>> "AGCTGCAGC".count("GC")`
`3`
	Return the number of occurrences of the substring `sub` in `S` (or `S[start:end]` if these arguments are given).

`S.endswith(sub[, start[, end]])`
 	
`>>> "abc".endswith("b",0,2)`
`True`
	Return true if `S` ends with the the substring `sub`, otherwise return false. Search `S[start:end]` if these arguments are given.

`S.find(sub[, start[, end]])`
 	
`>>> ">gryllus".find("gryll", 1)`
`1`
	Return the lowest index in `S` where the substring `sub` is found, starting and ending at indices `start` and `end` if these arguments are given. Return -1 if `sub` is not found.

`S.expandtabs([tabsize])`
 	
`>>> t = "  a"  # using a tab`
`>>> len(t); len(t.expandtabs())`
`2`
`9`
	Return a copy of `S` where all tab characters are expanded, i.e. replaced with spaces. If `tabsize` is not given, expand each tab with 8 spaces.

`S.isalnum()`
 	
`>>> "Don't add 5".isalnum()`
`False`
	Return true if all characters in `S` are alphanumeric (letters and digits) and there is at least one character, false otherwise.

`S.isalpha()`
 	
`>>> "GCAGCTTA".isalpha()`
`True`
	Return true if all characters in `S` are alphabetic (good old plain letters) and there is at least one character, false otherwise.

`S.isdigit()`
 	
`>>> "2+7=9".isdigit()`
`False`
	Return true if `S` has only digit characters, false otherwise.

`S.islower()`
 	
`>>> "amazing grace".islower()`
`True`
	Return true if all letters in `S` are lowercase and `S` contains at least one letter, false otherwise.

Figure 3.10: The most important string methods, called on some string `S`. Find more on `www.python.org` (search for string methods).

`S.isspace()`
 	
`>>> "\n \t ".isspace()`
`True`
	Return true if there are only whitespace characters (spaces, tabs, newlines) in `S` and `S` is not empty, false otherwise.

`S.isupper()`	Analogous to `islower`.

`S.join(seq)`
 	
`>>> "-".join(("a","b","c"))`
`'a-b-c'`
	Return a string which is the concatenation of the strings in the sequence `seq`, separated by copies of `S`.

`S.lower()`
 	
`>>> "STOP SHOUTING!".lower()`
`'stop shouting!'`
	Return a copy of `S` converted to lowercase.

`S.lstrip()`
 	
`>>> "  get to the point".lstrip()`
`'get to the point'`
	Return a copy of `S` with all whitespace at the left-most end removed.

`S.replace(old, new[, maxsplit])`
 	
`>>> "clmcncc".replace("c","a",3)`
`'almanac'`
	Return a copy of `S` with all occurrences of substring `old` replaced by `new`. If the optional argument `maxsplit` is given, only the first `maxsplit` occurrences are replaced.

`S.rfind(sub [,start [,end]])`	Same as `find` but search is performed from the right.
`S.rstrip()`	Analogous to `lstrip`.

`S.split([sep [,maxsplit]])`
 	
`>>> "a-bcd-ef-g-".split("-")`
`['a', 'bcd', 'ef', 'g', '']`
	Return a list of the substrings of `S` resulting from splitting it at each occurrence of `sep`. If maxsplit is given, at most maxsplit splits are done. If sep is not specified or `None`, any whitespace string is a separator.

`S.splitlines([keepends])`
 	
`>>> "one\ntwo\nthree".splitlines()`
`['one', 'two', 'three']`
	Return a list of the lines in `S`, splitting at newline characters. Newlines are not included in the resulting list unless `keepends` is given and true.

`S.startswith(sub[, start[, end]])`	Analogous to `endswith`.
`S.strip()`	Combines `lstrip` and `rstrip`.

`S.title()`
 	
`>>> "amazing gRace".title()`
`'Amazing Grace'`
	Return a title-cased version of `S` where all words start with uppercase characters and all remaining letters are lowercase.

`S.upper()`	Analogous to `lower`.

Figure 3.11: More string methods.

Chapter 4

Building a program

The control flow of a Python program is linear: It consists of a series of code blocks which are executed serially, one after the other. In this chapter, you'll learn about the basic mechanisms for designing a program's flow of control. These mechanisms are called *control structures*. First, however, I'll tell you what a code block is.

4.1 Code blocks and indentation

You've seen how to define a function: The keyword `def`, the function name, a colon, and the function body. So far in fact, you've only seen functions that could be defined all on one line. If the definition extends over one line, the function body should be put inside a *code block* beginning after the `def`, function name and colon — and even if it does fit on one line, most often your program will be more readable if you put the function body on a new line as a one-line code block.

A code block (also called a *suite*) is simply a list of lines of code which have been *indented*, all by the same number of spaces. Here are a few examples:

```
>>> def palindromize(s):
...   return s + s[::-1]
...
>>> def condition_factor(w, l):
...       temp = l*l*l
...       K = w/temp
...       return K
...
>>> palindromize("oprah se")
'oprah sees harpo'
>>> condition_factor(11000, 106)
0.0092358121133553194
```

The first function takes a string `s` as argument and returns a palindrome by joining together `s` and `s` reversed. The second function calculates the condition factor of a fish using its weight and length.

A *palindrome* is a word or sentence which spells the same if you read it backwards (ignoring spaces and punctuation).

The number of spaces you use to indent the block is up to you and your infallible sense of aesthetics. As long as all lines in the same block have the same indentation, Python is happy. All consecutive lines with the same indentation belong together — e.g. constituting the body of a function. The first line which is no longer indented as the one above it must match an indentation level of some block further up OR be at the outermost level. More on this later; you'll get the idea as we hop along.

The *condition factor* of a fish is a measure of its fatness calculated from its weight (without the gonads, mind you) in grams and its length in mm. (Source: `http://www.briancoad.com/Dictionary/C.htm`).

If you type in a function definition in the dynamic interpreter and then press the return key before giving the function body, you'll see three dots indicating that the interpreter now expects indented lines in a code block. When you're done typing the code block, simply enter an empty line. If you fail to indent, you'll get an error message saying `IndentationError: expected an indented block`.

4.2 The `if` control structure

Often you want something done only under certain circumstances, i.e. when some condition is true. That's implemented through the `if` control structure. Figure 4.1 uses a *flow chart* to illustrate how it looks in its simplest form. A flow chart is a diagram describing how control flows through some system. The flow of control follows the arrows, starting at the top-most circle. The diamond shapes are conditions: Several arrows may go out from condition diamonds, and if such an arrow is labelled with the truth value of the condition when evaluated under the current circumstances, control continues along that arrow. The square boxes are actions; each such boxed action is only executed if control flows through the box. The circles represent merging points where several possible paths through the diagram join.

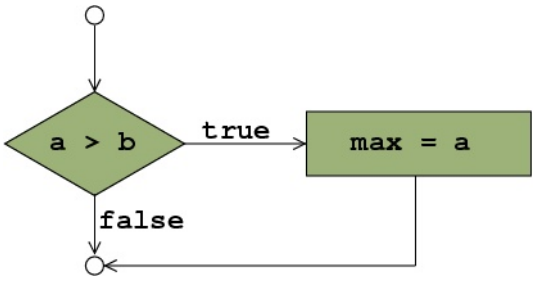


Figure 4.1: Flow chart illustrating the flow of control through an `if` statement.

The `if` is used like this:

```
if <condition>:
   <block of code>
elif <condition>:
   <block of code>
elif <condition>:
   ..
else:
   <block of code>
```

The `elif` and `else` parts are optional. A *condition* is any expression which is either true or false. The flow of control through an `if` control structure always begins with the first condition. If it is/evaluates to `True`, the first block of code is executed, and after that the program continues with the first statement after the whole `if` control structure. If the first condtion is `False`, the conditions of any `elif`'s (short for "else if") are evaluated in turn. If one of them is `True`, the corresponding block of code is executed and afterwards control continues after the control structure. If no conditions are `True`, the `else` block of code, if one exists, is executed. Figure 4.2 shows an `if/else` example through a flow chart.

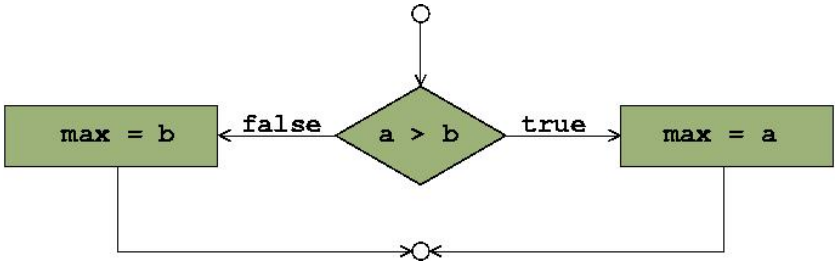


Figure 4.2: Flow chart illustrating the flow of control through an `if/else` statement.

A condition may be any comparison between variables and values using the relational operators <, >, <=, and >=, or the equality operators == and != (meaning $\neq$), but it may also be a function call which returns `True` or `False`. In Python, you use two ='s for equality testing to distinguish it from variable assignment. That may be a hard one to remember, but since the two operations are fundamentally different, it makes sense to use different notation for them. Instead of !=, you may also use <>. To illustrate:

```
>>> a = 8
>>> b = 2
>>> a == b
False
>>> a != b
True
```

```
>>> a == b*4
True
```

And some conditions using strings:

```
>>> s = "abcdEfg"
>>> s.islower()
False
>>> "A day in the life" < "Back in the USSR"
True
```

Note that you can use the relational and equality operators on strings too. A string s_1 is "less than" another string s_2 if s_1 comes before s_2 in an alphabetical ordering, like in the example above.

Figure 4.3 shows a program which tells you the category of a given intelligence quotient (IQ) score. The output from a test run is shown in Figure 4.4.

```
1  iq = int(raw_input("Enter IQ: "))
2  if (iq < 0) or (iq > 300):
3      print "Invalid value"
4  elif iq < 70:
5      print "Very low"
6  elif iq < 90:
7      print "Low"
8  elif iq < 110:
9      print "Normal"
10 elif iq < 130:
11     print "High"
12 else:
13     print "Very high"
```

Figure 4.3: Program `iq.py`.

```
1  Enter IQ: 107
2  Normal
```

Figure 4.4: Output from program `iq.py`.

In general, if you have some set of intervals and a variable, and you want to know which interval contains the variable, this is a neat way to do it. The program actually tests for the mutually exclusive intervals

0 – 69,
70 – 89,
90 – 109,
110 – 129, and
130 – 300.

The essential thing to remember is to perform the tests in the right order. (Try reversing the order of the tests in lines 4, 6, 8, and 10: Suddenly almost any IQ is deemed high and none are normal, low or very low. See why?)

Note the *logical operator* `or` in line 2 in Figure 4.3. The line means what it says: If one of the conditions is true, the value is invalid. There are three logical operators in Python: `or`, `and`, and `not`. They are used to combine conditions:

- A composite condition built from two inner conditions `or`'ed together (called a *disjunction*) is true if at least one of the inner conditions is true.
- If they are `and`'ed together (to form a *conjunction*), both of them have to be true in order for the conjunction to be true.
- A condition prefixed by `not` is called a *negation*. A negation is true if the inner condition is false, and vice versa.

Python evaluates the inner conditions of composite conditions left to right and doesn't go further than necessary to know the final result: If the left-most inner condition of a disjunction is true, the overall result will be true regardless of the other inner conditions, and so they are not evaluated. If the left-most inner condition of a conjunction is false, the conjunction is false regardless of any other inner conditions, and therefore they are not evaluated.

Actually, Python accepts variables and objects too in the place of conditions. Any non-zero object and any variable whose value is not 0 and not `None` count as `True`. Only 0 and `None` count as `False`:

```
>>> if 7:
...   print "True"
...
True
>>> n = None
>>> if n:
...   print "True"
... else:
...   print "False"
...
False
>>> s = "Cellar door"
>>> if s:
...   print "A string counts as True"
...
A string counts as True
```

```
>>> if not 0:
...   print "0 counts as false"
...
0 counts as false
```

4.3 The `while` control structure

How do you check if some sentence is a palindrome? Well, informally you check if it is spelled the same forwards and backwards. More formally, in a Python sort of setting, you could remove all spaces and punctuation from the sentence, turn all letters into lowercase, and check that the first character matches the last, the second matches the next-to-last, etc.

In fact, you would probably move one character at a time along the string, left to right, matching each character to a mirror partner in the other half, until you successfully reached the midpoint of the string. Now this type of behavior is called a *loop*: Doing the same thing over and over again until some condition becomes true. The `while` control structure implements it this type of loop; it is illustrated in Figure 4.5 using a flow chart.

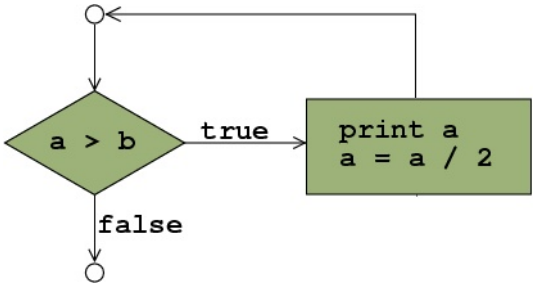


Figure 4.5: Flow chart illustrating the flow of control through a `while` statement. Control starts at the top-most circle, then goes on to evaluate the condition in the diamond-shape. If the condition is true, the code in the action box is evaluated, and then control goes back to reevaluate the condtion. This loop continues until the condition becomes false.

In Python, a `while` statement looks like this:

```
while <condition>:
   <block of code>
```

The keyword `while` is followed by a condition, then a colon, and then, suitably indented, a block of code (called the *body* of the loop) which will be executed if and only if the condition evaluated to true. If not, the loop body is skipped and the control flow continues with the first statement after the loop body. You cunning rascal of course already realize at this point that unless you actually do something inside the loop body which will change the outcome of

some future evaluation of the loop condition to make it false, the loop will go on indefinitely. You'll be surprised how often you'll forget to anyway.

Back to the palindrome checking problem; for now, let's not consider strings with punctuation. The program in Figure 4.6 presents a solution.

```
1  s = raw_input("Sentence? ").replace(" ", "").lower()
2  i = 0
3  ispalindrome = True
4  while i < len(s)/2:
5      if s[i] != s[-1-i]:
6          ispalindrome = False
7          break
8      i += 1
9  if ispalindrome:
10     print "Your sentence is a palindrome"
11 else:
12     print "Not a palindrome:",
13     print "%s doesn't match %s"%(s[i], s[-1-i])
```

Figure 4.6: Program `ispalindrome.py`.

Line 1 gets a sentence from the user. The output from `raw_input` is handed directly to the string method `replace` which replaces all spaces with nothing — essentially removing them — before handing the now space-less string to another string method, `lower`, which turns all uppercase letters into lowercase. Nice pipeline, eh?

In the loop, we'll look at the i'th character from the left and match it to the i'th character from the right. Our index counter is `i`, initialized in line 2, and further we'll need a status variable, `ispalindrome`, which we'll set if `s` turns out not to be a palindrome. Such a variable is sometimes called a *flag* which is "raised" if some particular event has happened.

In the `while` loop, we should keep going until we hit the middle of the string — no need to go further since we've already matched the second half to the first half when we get to the middle. Thus, the loop condition in line 4 is `i < len(s)/2`.

Inside the loop, properly indented, is an `if` statement: If the character in index `i` does *not* match the one in index `(-1-i)` (recall that strings can be indexed backwards with negative numbers), `s` is not a palindrome, and we set the `ispalindrome` to `False`. And then we use the `break` command to leave the loop; no need to waste more time if we already know that the sentence is not palindromic. `break` causes Python to leave the (innermost, in case several loops are nested inside each other) loop and continue with the first statement after the loop body.

The last statement in the loop body is in line 8 where the index variable `i` is

incremented. If we had forgotten to do that, the loop would never terminate. And note here the potential consequences of a wrong indentation: If line 8 was erroneously indented in line with the `break` of line 7, it would be seen by Python to belong to the same code block as line 7. Consequently, `i` would not be incremented in case the character pair matched. A string like "Bob" would cause an infinite loop since the program would never move beyond the first character. You'll agree that every day holds the promise of new dangers.

When the loop terminates, if `ispalindrome` is still true, all characters matched their mirror in the other half and `s` is a palindrome. Otherwise, there was a mismatch somewhere. See the program output in Figure 4.7, and then let's move on to another example.

```
1 Sentence? Never odd or even
2 Your sentence is a palindrome
3
4 Sentence? Alabama
5 Not a palindrome: l doesn't match m
```

Figure 4.7: Output from program `ispalindrome.py`.

Imagine that the string `g` is a DNA sequence from some person's genome. You want to check whether this piece of DNA contains the trinucleotide repeat `CAG` and, if it does, with what multiplicity. E.g. the repeat `CAGCAGCAG` has multiplicity 3 since the pattern is repeated 3 times. In other words, the job is to go through `g` and find the maximal number of consecutive occurrences of the pattern `CAG`.

With pen and paper, you'd probably scan the DNA string left to right to spot any `CAG` repeats, remembering the multiplicity of the longest one seen so far. In fact, you'd be doing the same thing over and over again for the entire length of the string. Something along these lines, perhaps:

A *genome* is the total collection of genetic material of some living organism. All genomes are organized in a set of *chromosomes* encoded as *DNA* (Deoxyribo-Nucleic Acid). DNA are gigantic molecules built out of smaller molecules called *nucleotides* chained together. Much of the DNA of any genome consists of repeated material. A *trinucleotide repeat* is a pattern of three nucleotides repeated shoulder to shoulder some number of times. Humans have a specific occurrence of the pattern CAG with 4 to 20 repetitions on chromosome 19, but a *mutation* (a random genetic modification) has created extra repetitions beyond 20 in some humans. This unfortunate mutation causes the disease Pure Spinocerebellar Ataxia which impedes walking and speech. (Source: `http://www.neuro.wustl.edu/neuromuscular/ataxia/domatax.html\#6`).

While I'm not done, search the string left to right:

1. If I don't see a `CAG`, move one character down the string.

2. If I see a `CAG`, I'm in a pattern repeat. Add one to the multiplicy of this current repeat and check if it beats the current max.

```
1  # NB. The human chromosome 19 is about 80.000 times the
2  length of this string

3  g = """CAGTACTACCTCAGACGTCAGCATCAGTACATCGATCGATCGATGCTAGCT
4  AGCGCATCGATCTACGTACGTGACTGATCGTACGTACGACATCGTCCAGCAGCAGCAG
5  CAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGC
6  AGTACGTACGTATCATCTATTAGCGCGCGTAATCCCGATATCTTATACTACTCAGGAT
7  GCAGCAGCAGCAGCAGCAGCAGTGG"""

8  pointer = 0 # index pointer into g
9  mul = 0     # multiplicity of current pattern match
10 maxmul = 0  # max multiplicity of any pattern match found

11 while pointer < len(g):

12     if not g[pointer:].startswith("CAG"):
13         mul = 0
14         pointer += 1
15     else:
16         mul += 1
17         pointer += 3
18         if mul > maxmul:
19             maxmul = mul

20 if maxmul > 0:
21     print "Pattern found with max multiplicity", maxmul
```

Figure 4.8: Program `trinucleotide.py`.

The program in Figure 4.8 realizes the algorithm crudely sketched above. In lines 8–10, we create and initialize a few variables we'll need: In each round of the loop, we'll look at a substring of `g` starting at some index; `pointer` will be this index. At all times, we'll either be "inside a current repeat", or not inside one. The variable `mul` will be the multiplicity of this current repeat — if we're not currently looking at one, we'll set `mul` to 0. And finally, `maxmul` will hold the maximal multiplicity we have seen in any repeat so far; it's initialized to 0.

The loop should continue as long as we haven't looked at all of `g`; i.e. as long as the `pointer` is less than the length of `g` (line 11).

Inside the loop in line 12, we check if `g[pointer:]`, the substring we're looking at, starts with `CAG` (remember that you can use string methods like `startswith` on any string, including one resulting from a slice operation). If this substring does *not* start with `CAG`, we reset `mul`: That means we're not inside a pattern repeat, and so the current multiplicity should be set to 0. We also increment `pointer` to move one character further down in `g` (the *augmented assignment symbol* `+=` in an expression `x+=y` is shorthand for `x=x+y`. You may also use `-=`, `*=`, `/=`, `**=` and `%=`).

If the substring does start with `CAG`, the program will enter the `else` branch

in line 15. In that case, we're either starting a new current repeat or we're continuing one. In both cases we should increment the multiplicity of the current repeat, `mul`. Next, we add 3 to `pointer` in line 17 to move past both the `C`, the `A` and the `G` before the next round of the loop. And finally we check whether the new, updated `mul` is greater than the max seen so far. If so, we update `maxmul` and set it to `mul` (lines 18–19).

When the loop terminates, `pointer` has moved all the way from left to right through `g`, and therefore we're sure to have visited all repeats. If `maxmul > 0`, we've found at least one, and we print a triumphant message. See the program output in Figure 4.9.

```
1 Pattern found with max multiplicity 24
```

Figure 4.9: Output from program `trinucleotide.py`.

The programs in Figures 4.6 and 4.8 are the first examples of genuine algorithm implementations. The algorithms may not be completely obvious to you; make sure you see why the programs work before you move on. Most often there are more than one way of solving a given problem — more than one algorithm. We'll return to the issue in the last part of this chapter.

As a final note, notice the occasional empty lines in the program code. You should make it a habit to include a lot of empty lines in your code since it increases readability immensely. Code is usually neither particularly visually pleasing nor immediately understandable to read, but it helps a lot if you add empty lines. This way you can emphasize the logical structure of your program, e.g. by grouping a few coherent lines together, separating function definitions from each other and from the main part of the program, etc. Also, adding such spatiality to large programs makes it a lot easier for you to scroll around in them and find some specific code block you're looking for – your brain will remember the "contour" of it.

4.4 The `for` control structure

The `for` control structure resembles the `while` in the sense that it's also a loop, but here the loop is carried out in sort of a more strict way. You supply a sequence of items and then the `for` loop body is executed once for each item, in order left to right. What happens is that a temporary "counter" is used to iterate over the sequence. If we call the counter x and the sequence y, in each round of the `for` loop, x is assigned the next item from y until all have been processed. Informally, a `for` loop looks like this:

for x in y:
    <do something>

Figure 4.10 illustrates a `for` loop using a flow chart.

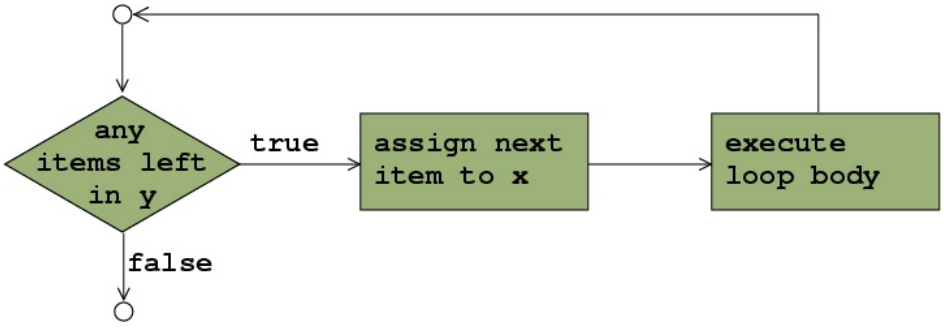


Figure 4.10: Flow chart illustrating the flow of control through a `for` statement. Note that it is not illegal to supply an empty sequence `y`. If `y` is empty, the loop body is just never entered.

In Python you use a `for` loop like this (the counter variable may have any name you choose):

```
for <counter> in <sequence>:
   <block of code>
```

I haven't told you formally about sequences yet and I won't until Chapter 6, but you may have noticed some in the string method tables (Figures 3.10 and 3.11). Briefly, a sequence is either a *list* or a *tuple* of values. Lists are written with square brackets, tuples with parentheses. Both types of sequence are indexed just like strings:

```
>>> l = [1,2,3]
>>> t = ("any", "kind", "of", "values")
>>> l[0]
1
>>> t[-1]
'values'
>>> t[1:3]
('kind', 'of')
```

In fact, a `for` loop just needs something that *evaluates to* a sequence, not necessarily an explicitly written one. E.g., you could call a function which returns a sequence, and of course you could use a variable pointing to a sequence. Figure 4.11 shows a little program demonstrating both the `for` loop and one of the grand old men of string methods.

The output is shown in Figure 4.12. The method `split` splits `s` by each whitespace character returning a list of words. Such a list is perfect for a `for` loop, and to go through it you only need to come up with an appropriate name for a counter variable. Here, my inspired choice is `word`.

The loop then begins with the variable `word` being assigned the first item in the list which is the string `One`. Inside the loop body, I simply print `word`.

```
s = "One flew over the cuckoo's nest"
for word in s.split():
    print word
```

Figure 4.11: Program `for.py`.

After the loop body terminates, Python checks if there are more items in the list. If so, the next item is assigned to the counter variable, and another turn in the loop body is taken. If not, the first statement after the `for` control structure, if any, is executed.

```
One
flew
over
the
cuckoo's
nest
```

Figure 4.12: Output from program `for.py`.

You can go through the individual characters of a string using a `for` loop:

```
>>> for c in "Spread the word!":
...  print c,
...
S p r e a d   t h e   w o r d !
```

The inconspicuous keyword `in` can also be used to check if some value is present in a sequence or a string if you don't care about the index of any match you might find. Using `in` is more readable than using `find`:

```
>>> s = "See you next Wednesday"
>>> if s.find("next") > -1: print "Found it"
...
Found it
>>> if "next" in s: print "Found it"
...
Found it
```

If you have a sequence of items and you need to do something for most of them while you simply ignore others, you may need the `continue` command.

With `break`, you break out of the current surrounding loop and never finish it, but with `continue` you skip the remainder of the loop body *but continue with the next round of the loop*, if any rounds remain. See the program in Figure 4.13 which encrypts a text into Pop Cipher. Figure 4.14 shows a test run.

```
1  text = raw_input("Input text: ")
2  vowels = ['a', 'e', 'i', 'o', 'u', 'y' ]
3  for c in text.lower():
4      if not c.isalpha():
5          continue  # skip non-letters
6      if c in vowels:
7          print c,    # just print vowels
8          continue
9      # At this point we know that c is a consonant
10     print "%so%s"%(c, c),
```

Figure 4.13: Program `popify.py`.

Line 2 defines a list holding all vowels. In line 3, I turn the text into lowercase and enter a loop to go through all its characters. Since I intend to ignore characters which are not letters (such as numbers and punctuation signs), I do a check in line 4 (using the string method `isalpha`). If the current character is not a letter, I leave the loop body but `continue` with the rest of the loop. If I had used `break`, I would have exited the loop completely and any text beyond the special character would have been ignored.

The *Pop Cipher* is a simple text encryption technique introduced by Astrid Lindgren in her children's book about master detective Kalle Blomkvist. Each consonant in the text is turned into a syllable by doubling it and adding an o in the middle; vowels are left untouched.

Vowels should just be printed without further ado, so I perform a check by comparing with my pre-assembled list in line 6, print it if I found one, and then `continue` with the rest of the text. If I get all the way to line 9, I have a character which is not a non-letter and not a vowel — i.e., a consonant. So I wrap two copies of it around an o (using string formatting) and print.

```
1  Input text: The light was yellow, sir.
2  tot hoh e lol i gog hoh tot wow a sos y e lol lol o wow
3  sos i ror
```

Figure 4.14: Output from program `popify.py`.

4.5 Example including sort of everything you've seen so far

To (almost) conclude this chapter, here's a small guessing game program. You think of a number, and the program guesses it by asking "higher or lower?" questions after each wrong guess. The algorithm keeps track of an interval in which it knows your number is contained. After each wrong guess, it reduces this interval: If the guess was too high, it knows your number must be in the subinterval containing numbers *less than* the wrong guess. If the guess was too low, the right number must be in the other subinterval. So, for the next round, only the relevant subinterval is kept for further consideration. To make sure as much of the interval as possible is thrown out in each round, the guess the program makes is exactly the mid point of the current interval. That way, the interval size, and thus the number of remaining possibilities, is halved in each round. The program is shown in Figure 4.15, output from a test run in Figure 4.16.

Lines 4–10 are initializations: `attempt` counts how many attempts the program needs to guess the number, `low` and `high` define the end points of the current interval, and `got_it` is a status variable: Its value tells the program when to stop. It is initialized to `'n'` signifying that the program hasn't got the right number yet. Lines 7 and 8 define the questions the program will ask in each round. Each question has a conversion specifyer in it which is to be "filled out" by a variable in each round.

The algorithm consists of a `while` loop (line 11) which runs until the program guesses the number, i.e. as long as `got_it` is `'n'` That's good programming ethics: A loop condition should clearly state when the loop should terminate. (Thus, using things like `while True` and then a `break` inside the loop somewhere is often considered bad conduct because it's not immediately clear from the loop condition why the loop terminated).

Inside the loop body, the mid point of the interval is calculated and stored in the variable `guess` (line 12) which is then substituted in the first question to the user (line 13). The user then answers 'y' or 'n'. If the guess was correct, `got_it` is set to `'y'` (line 14) and the rest of the loop body is skipped with `continue`. The subsequent evaluation of the loop condition then becomes false and the loop terminates.

If the guess was incorrect, the next question is posed, again substituting this round's guess (line 16). Finally, in lines 17–21 the interval end points are updated according to the user's answer to the second question, and `attempt` is incremented. And in the end when the word has been guessed and the loop is abandoned, `attempt` is reported.

I'd like to stress again the importance of using the correct indentation: It's essential to the correctness of your program. Sometimes a wrong indentation is caught by the interpreter so you'll be told about it..

```
>>> if 0:
...     print "0 counts as True"
```

```
raw_input( """
Think of a number between 0 and 100 and I'll guess it.
Press return when you're ready.""" )

attempt = 1
low = 0
high = 100
question1 = "\nIs your number %d (n/y)? "
question2 = "Is your number higher or lower than %d (h/l)
? "
got_it = 'n'

while got_it == 'n':

    guess = (high + low) / 2
    got_it = raw_input( question1%guess )

    if got_it == 'y':
        continue

    answer = raw_input( question2%guess )

    if answer == 'h':
        low = guess
    elif answer == 'l':
        high = guess

    attempt += 1

print "I guessed your number in %d attempts"%attempt
```

Figure 4.15: Program `i_guess_your_number.py`.

```
...  else:
  File "<stdin>", line 3
    else:
       ^
IndentationError: unindent does not match any outer
indentation level
```

.. but sometimes, it's not a syntactical error but rather a "logic error", i.e. an error which causes your program to misbehave without you realizing it. This is a very bad thing. Lo and behold this sneaky pitfall:

```
>>> cousin_Daisy_is_home = True
>>> aunt_Irmgard_is_home = False
>>> if cousin_Daisy_is_home:
```

```
Think of a number between 0 and 100 and I'll guess it.
Press return when you're ready.

Is your number 50 (n/y)? n
Is your number higher or lower than 50 (h/l)? l

Is your number 25 (n/y)? n
Is your number higher or lower than 25 (h/l)? h

Is your number 37 (n/y)? n
Is your number higher or lower than 37 (h/l)? h

Is your number 43 (n/y)? n
Is your number higher or lower than 43 (h/l)? h

Is your number 46 (n/y)? y
I guessed your number in 5 attempts
```

Figure 4.16: Output from program `i_guess_your_number.py`.

```
...     if aunt_Irmgard_is_home:
...         print "Visit cousin Daisy and aunt Irmgard"
...     else:
...         print "Cousin Daisy is not home :{"
...
Cousin Daisy is not home :{
```

The `else` is on the same indentation level as the `if aunt_Irmgard_is_home` block, but in fact it logically belongs on the outermost level together with `if cousin_Daisy_is_home`. The code shown and hence the conclusion printed by the program is actually wrong — cousin Daisy *is* home, the situation is perfect!

In other words, when dealing with `if`'s and especially `else`'s and in general any indented code blocks, beware, be-very-ware. You don't want to wake up one day and realize that cousin Daisy was home and you never found out about it because you messed up with an `else`. She won't understand.

4.6 Extensions to `for` and `while`

We might want to introduce a maximum number of guesses allowed, say 5, in the guessing game. One way would be to insert this code between lines 15 and 16 (using `break` as an *exceptional* way of leaving the loop, not the "intended" way which is formulated in the loop condition):

```
if attempt == 5:
    break
```

Then the loop would run until the program guessed the user's number but at most five times. The question then becomes how to decide *why* the loop ended: For the first or the second reason?

Sometimes you want something done after a loop only if it terminated normally and not exceptionally with a `break`. The obvious solution to such a situation is to use this variation of the `while` control structure:

```
while <condition>:
   <block of code>
else:
   <block of code>
```

The naming here is awful because the `else` block is evaluated only if the `while` loop terminates *normally*; if a `break` causes the loop exit, the `else` part is ignored. So if everything goes well, you go to the `else` afterwards. If you're forced out, don't. But so be it, it's a handy construction. Let's make a further modification to the guessing game program and move the final `print` statement inside an `else`. The relevant part of the program now looks like this:

```
while got_it == 'n':
    ..

    if got_it == 'y':
        continue

    if attempt == 5:
        break
    ..

else:
    print "I guessed your number in %d attempts"%attempt
```

With this construction, the tadaa! `print` statement is only executed if the loop terminated in the intented way through `got_it` becoming `'y'`. Thus, if the program didn't actually guess the number within the allowed number of guesses, it doesn't tadaa anyway like a complete fool.

The `for` control structure comes in an analogously extended version.

4.7 A brief note on algorithms

As I mentioned there are usually many ways to solve a given problem. Which algorithm to choose depends on which qualities you're looking for in your solution. Do you want your program to be fast? Do you want it to be easily understandable? Do you want it to use as little memory as possible for variables etc.? Also, one thing is the algorithm, another is how to express and

implement it in Python code. You'll see and hear more on algorithms later, but as a small illustration of how implementation issues, we might revisit one of the problems of this chapter.

Consider again the problem of deciding whether a given string is a palindrome (see Figure 4.6). Once the input string was stripped of spaces and lowercase'd, the algorithm was to meticulously compare each character in the left half with its right-half mirror character. This approach took some accounting in terms of using and keeping track of indices. The *algorithm* is obviously correct, but it's less obvious that its *implementation* is, too. Do we remember to check the middle letters? Does index i really mirror index $-1-i$?

Figure 4.17 shows an alternative solution. Which one do you prefer?

```
1 s = raw_input("Sentence? ").replace(" ", "").lower()
2 if s == s[::-1]:
3     print "Your sentence is a palindrome"
4 else:
5     print "Not a palindrome"
```

Figure 4.17: Program `ispalindrome2.py`.

Recall that `s[::-1]` yields `s` reversed. When Python compares two strings, it must somehow compare them character by character, so the *underlying* algorithm of this program, `ispalindrome2.py`, still compares characters one by one just as `ispalindrome.py` did. The programs differ in three ways, though:

- It is *much* easier to convince yourself that `ispalindrome2.py` works correctly.
- Along the way, `ispalindrome2.py` *copies* (and reverses) the input string.
- Whereas `ispalindrome.py` only compared the first *half* of the input string `s` with the other half, `ispalindrome2.py` compares the *whole* of `s` with its reversed copy.

The first point is of course a huge advantage. The second point means that `ispalindrome2.py` uses twice the memory of `ispalindrome.py` — plus it spends time reversing `s`. And finally, the third point means that the loop of `ispalindrome2.py` is likely to take twice as long as the loop of `ispalindrome.py`.

Now for toy problems like this one where the data you are working with is small, space and time issues are of no importance whatsoever, and you're better off with a solution which is easy to follow and implement correctly. The time you save not having to debug the more efficient solution easily pays for the extra milliseconds your easy-to-implement program lasts. This especially

applies to programs which you'll only ever run once or very few times; the time you spend writing and debugging a program counts too.

However, in many cases efficiency is more important. When working with very large data, an algorithm which takes twice as long as another, or needs to copy the entire data set, may be unacceptable. So the take-home message here is that it's always a good thing to consider alternative algorithms and implementations — but that you should include the program's data in these considerations. Sometimes razor-sharp efficiency is essential; often clarity should take precedence.

Now you know about:

- Indentation
- Augmented assignment symbols
- `if/elif/else`
- `while/else`
- `for/else`
- `and`, `or`, `not`
- `break`, `continue`

Chapter 5

Modules, functions, namespaces and scopes

We've touched upon the concept of local variables back in Chapter 2: The parameter names of a function are local to that function and cannot be used outside the function body. This is an example of a *scope*: The scope of a variable, or any identifier, is the portions of the program where it can be used; where it is "visible". A *namespace* can be regarded as the complement to a scope: The scope defines where a given identifier is visible, whereas a namespace defines which identifiers are visible at a given point in the program.

Your namespaces are altered when you create new functions, variables, etc., and when you import identifiers from external modules. What exactly happens when you use the `import` command is the topic of Section 5.2 where you'll see some examples, but before that I'll introduce the concept of character codes.

5.1 Character codes

Each character has a corresponding integer number, called its *character code* or *ordinance number*. E.g., the character `a` is number 97 and `!` is number 33. Python has two built-in functions which translate between characters and character codes called `ord` and `chr`. The latter takes a number between 0 and 255 and returns the character with that character code; the first takes a character (a one-letter string) and returns its character code. The program in Figure 5.1 creates a Caesar Cipher using these functions.

A Caesar Cipher is a simple text encryption technique which simply shifts the entire alphabet a fixed number of positions. E.g., with a shift of three, all a's become d's, all b's become e's, and so on. It "wraps around" the end of the alphabet so that x'es become a's and y's become b's, etc. If you shift some text *n* positions, the encrypted text looks like garbage, but of course you only have to switch the encrypted text *-n* positions to recover the original text.

The *Caesar Cipher* was invented by the famous Roman Julius Caesar (ca. 100 BC – 44 BC) who used it to communicate with his generals. (Source: `http://en.wikipedia.org/wiki/Caesar_cipher`).

In the good old English alphabet, there are 26 letters. Becoming computer scientists, let's number them from 0 to 25. There are two tricky issues in doing the Caesar Cipher: converting from the 0 to 25 range to the character codes Python uses, and handling the wrap-around so that we can switch characters off the edges of the interval and reinsert them in the opposite side.

In the Unicode character set which Python uses, the lowercase letters are numbered from 97 to 122, and the uppercase letters from 65 to 90. If we subtract 97 from the character code of a lowercase letter, we therefore get the letter's rank in the alphabet, i.e. a number between 0 and 25. We can then shift this number by adding some constant to it. The result may "overflow" beyond 25 (or "underflow" below 0 with a negative shift), but we can calculate the result modulo 26 to ensure that is stays between 0 and 25. Finally we can add the 97 back again to get to a proper lowercase character code.

Say we want to switch the character `y` 6 positions. The character code of `y` is 121. Subtract 97 and you get 24 — the alphabetical rank of `y` if you start from 0. Add 6 and you have 30 which is beyond 25, but 30 modulo 26 is 4. The letter with alphabetical index 4 is `e`, and to get its character code, we add 97 and get 101.

That's the deal, and that's what goes on in the function `shift_char` in Figure 5.1 (lines 1–7). It takes two arguments: A character `c` and a number of positions to shift, `shift`. It uses string methods to check whether the character is a lowercase or uppercase letter (in case it is neither, it is returned untouched in line 7). The initial conversion from character to character code is handled by `ord`, while the final conversion from the shifted character code back to a character is handled by `chr`.

```
1  def shift_char(c, shift):
2      if c.islower():
3          return chr(((ord(c)-97+shift)%26)+97)
4      elif c.isupper():
5          return chr(((ord(c)-65+shift)%26)+65)
6      else:
7          return c # don't shift non-letters

8  def caesar_cipher(s, shift):
9      m = ""
10     for c in s:
11         m += shift_char(c, shift)
12     return m

13 text = "Alia iacta est!"  # the die has been cast
14 print text, "becomes", caesar_cipher(text, 6)
```

Figure 5.1: Program `caesar_cipher.py`.

To cipher a complete text, each single character needs to be shifted. That's the job of the program's second function, `caesar_cipher` (lines 8–12). It creates a new message string `m` to hold the encrypted result, and it then iterates through the characters of its input string `s`, shifting each using the `shift_char` function and adding it to `m`.

The main part of the program creates a text and prints out the encrypted version by calling `caesar_cipher` with the extra `shift` argument 6:

```
Alea iacta est! becomes Grog ogizg kyz!
```

Note how the uppercase A correctly becomes an uppercase G, and that the ! is left unchanged. The output is a bit confusing, though; the "becomes" in the middle makes it a little unclear what is what. The cool of it is all lost. Let's do something about it.

5.2 Reusing previously written code

If your program prints a lot of output to your screen, it might help if some of the output could be printed in a different color, say red. That's possible using a magic spell (an "ANSI terminal control escape sequence"!) before and after the text. You need to print the Escape key, then a special "start red" code, then the text, then the Escape key again, and finally a "reset color" code. You can't just press the Escape key to insert it in a string, though, your editor probably treats any Escape key presses in some special way. Fortunately, it discretely has a character code: 27. Try this..

```
>>> esc = chr(27)
>>> print "%s[31;2mInternationale%s[0m"%(esc, esc)
```

.. and you'll get:

```
Internationale
```

This feature would brighten up the output of the Caesar Cipher program — and most probably heaps of other programs to come. However, the secretive color codes and the Escape key character code and all that is horrible; you don't want to have to remember that each time you want some red text. Instead, let's create a function with some nice, sensible name which hides the dirty details. Figure 5.2 shows one, in a program called `stringcolor.py`. It simply wraps the given string in the required Abracadabra and returns it. It doesn't print it itself — who knows if the user wants the newline right after the string.

```
1 def inred( s ):
2     return "%s[31;2m%s%s[0m"%(chr(27), s, chr(27))
```

Figure 5.2: Program `stringcolor.py`.

Let's spice up the Caesae Cipher program using the new function. Assuming the two programs are located in the same directory (I'll explain why later), we can modify the cipher program so that it imports the `inred` function as shown in Figure 5.3.

```
1  import stringcolor

2  def shift_char(c, shift):
3      if c.islower():
4          return chr(((ord(c)-97+shift)%26)+97)
5      elif c.isupper():
6          return chr(((ord(c)-65+shift)%26)+65)
7      else:
8          return c # don't shift non-letters

9  def caesar_cipher(s, shift):
10     m = ""
11     for c in s:
12         m += shift_char(c, shift)
13     return m

14 text = "Alia iacta est!"  # the die has been cast
15 cipher = stringcolor.inred(caesar_cipher(text, 6))
16 print stringcolor.inred(text), "becomes", cipher
```

Figure 5.3: Program `caesar_cipher2.py`. The only changes compared to Figure 5.1 are in lines 1 and 14–16.

In line 1, Python is told to import the file `stringcolor.py` — note that the `.py` extension must be left out. The `import` command means "execute all the code in the following module file". The result is that the `inred` function is defined for the mother program to use; it becomes accessible through the new identifier `stringcolor` which is created by the `import` statement to point to the module.

In lines 15 and 16, the `inred` function is called: `stringcolor.inred` means "look inside the module `stringcolor` for an identifier called `inred`". The *dot operator* in between the names is what performs the look inside. Running the program gives you:

```
Alea iacta est!  becomes Grog ogizg kyz!
```

Now, a Caesar's Cipher program is obviously one of *the* most useful programs one might ever think of. Ever so often we'll need it, and, as you've seen, we should probably simply import it:

```
>>> newstring = "Veni, vidi, vici"
>>> import caesar_cipher2
Alea iacta est! becomes Grog ogizg kyz!
>>>
```

Alas, the main part of the `caesar_cipher2.py` program is executed when we import it, even though we set out to encrypt another string. We don't want these old remnants! As I mentioned earlier, importing a module really means to execute all code in it, including the stuff outside function definitions etc. So, in order to make it generally useful, we should cut out the main program (lines 14–16 in Figure 5.3) and leave only the functions and the `import` statement.

In summary, if you are writing some useful code which you realize you'll be reusing again in the future, you should put it in a separate file (perhaps together with related functions) which contains no main program. Thus, if you `python`'ed such a file directly, apparently nothing would happen since the functions defined in it are not *called* anywhere, but when you then need the functions, you simply import the file module from another program.

5.3 Modules and ways to import them

When Python sees an `import` statement, it first looks for the module file in the same directory where the program currently running is located. Then it looks in the directory where the Python language installation houses its library of modules (probably `/usr/lib/python2.3` or something similar). If it's not found there either, Python needs to be told where to look for it. From within a Python program, you need to add the path where it should look to a list called `path` which is found in the module `sys`. Thus, if you create a module file in the directory `/users/<username>/pythonmodules/`, you should insert these lines before attempting to import it from somewhere outside the directory:

```
>>> import sys
>>> sys.path.append("/users/<username>/pythonmodules")
```

Note that you should not use the shorthand ~ for `/users`; your operative system might get confused.

You can also tell your operating system where Python should look for your modules rather than tell Python directly. That is done by means of a so-called *environment variable* called `PYTHONPATH`. Usually you'd prefer it to be set by

the operating system when you log in so that Python automatically looks in the right place without you having to tell it explicitly in every program. How to do that is beyond the scope of this text, however.

Python comes with a large library of *modules* — see `http://www.python.org/doc/current/modindex.html`. A module contains mutually related "extra" functionality in the shape of function definitions, constants, etc. Thus, in many cases you don't have to write your own function to perform some task since the nice Python people have already done so, and chances are that they have done a better job in terms of efficiency, speed (and correctness!) than you will. It's a good thing to know your Python modules: Why waste time creating (first writing, then debugging..) code which the pros have already written? (To learn Python, yes yes, but go on to write the rest of your program which someone else hasn't already done for you).

The way to get access to all the benefits of some module is of course to import it. Say that you need to flip a coin in order to make some decision. How do you flip a coin in Python? You use the `random` module. This module contains a number of functions relating to random numbers. To see what you get when you import it, you can use our good old friend `dir` in the dynamic interpreter. Here's some of its output:

```
>>> import random
>>> dir(random)
['BPF', 'LOG4', 'NV_MAGICCONST', 'Random',
 ..
'paretovariate', 'randint', 'random', 'randrange',
'sample', 'seed', 'setstate', 'shuffle', 'stdgamma',
'uniform', 'vonmisesvariate', 'weibullvariate']
```

There's a few weird constants and a load of functions. One is called `randint`; it takes two integer arguments a and b and returns a random integer N such that $a \leq N \leq b$ (note that, for once, the end point b is *included* in the range of possible values). Thus, using the dot operator introduced in the previous section, you can flip a virtual coin (with sides numbered 1 and 2) like so:

```
>>> import random
>>> random.randint(1, 2)
1
```

We'll return to this wonderful module on numerous occasions. Besides being necessary for some purposes, randomness is just lovely in itself.

As we go along, we'll meet many of Python's other nice modules. They can all be imported like above, and the way to utilize some identifier `y` inside a module `x` is through the dot operator with `x.y`.

If some Python module — or some program of your own which you import — has a very long name, you might get tired of having to use it with the dot operator as the prefix to the identifier you're actually after. There are two ways to circumvent it. The first way is to *rename* the imported module using the `import/as` construct. For instance, you might rename the `stringcolor` module to just `sc` like this:

```
>>> import stringcolor as sc
>>> print sc.inred(''Warren Beatty'')
```

Nothing has changed except the name of the module. Its function is still called `inred` as you can see; the result of the function call above is, as expected:

```
Warren Beatty
```

The other way of avoiding having to use a long module name as prefix every time you access its content is more radical. Rather than just creating *one* new identifier named after the module which then becomes your only gateway to the goodies inside, you can instead *directly* import all the identifiers you need from the module using the `from/import` construct. E.g., to import the `randint` function from the `random` module, you could do this:

```
>>> from random import randint
>>> randint(1, 2)
2
```

The difference is that now you *only* have access to the `randint` function; everything else in the `random` module is out of your reach since you no longer have a handle to them (although all the code in the module is still executed, so you don't save any execution time by only importing the stuff you need). Moreover, some would say that this approach may obscure your program since it is no longer immediately visible from where the function originates. Still, you could go one step further down this road and rename the imported function to something more precise or more brief, e.g.:

```
>>> from random import randint as random_integer
>>> random_integer(10, 20)
13
```

You've already seen in Section 2.2 how one can import *all* the functions from some module with the `from/import *` construct: The `*` means "anything". Then it was the `math` module, and the statement was `from math import *`. In general, this approach cannot be recommended, though. Imagine you are a captain on a spaceship; then, evidently, you would need a function for adding entries to the *captain's log*. To calculate the starship's current position, you might also need some help from the `math` module, and so you might try something like what's shown in Figure 5.4:

```
1  >>> def log(item):
2  ...  return "Captain's Log, Stardate 44429.6: %s"%item
3  ...
4  >>> print log("We're on a mapping survey near the
5  Cardassian sector.")
6  Captain's Log, Stardate 44429.6: We're on a mapping
7  survey near the Cardassian sector.
8  >>>
9  >>> from math import *
10 >>> print pow(2.7, 9.4)
11 11345.3887894
12 >>>
13 >>> print log("The treaty ended the long conflict
14 between the Federation and Cardassia.")
15 Traceback (most recent call last):
16   File "<stdin>", line 1, in ?
17 TypeError: a float is required
```

Figure 5.4: What might go wrong when using `from <module> import *`.

Oops, what's this? In lines 1 and 2, we define a (very silly) `log` function which prints its string argument with a fancy prefix. When testing it in line 4, it works fine. Next, in line 9 we carelessly do the `from math import *` thingy which also works fine when we use its power function `pow` in line 10. Then when we intend to print the next log entry in line 13, suddenly Python complains that "a float is required". No it isn't, I wrote the `log` function myself! Well yes, but unfortunately we got more than we bargained for when we imported *anything* in the `math` module. There's a logarithm function in it called.. well guess what? Precisely: `log`! And this function *shadows* our own when it is brutally imported. The name `log` no longer points to our function which becomes forever more inaccessible.

So forget about the `from <somemodule> import *` statement. It's usually not a clever move. Instead, either import the whole module with

```
import <somemodule>
```

or import specifically what you need from it with

```
from <somemodule> import <function1>, <function2>
```

separating the identifiers you need with commas. To remedy the above shadowing problem, either do

```
import math
```

since then the logarithm function will be called `math.log`, or

```
from math import pow
```

since then the logarithm function isn't imported at all. If it turns out you also need the logarithm function, you might change its name while still keeping the name *pow*:

```
from math import log as logarithm, pow
```

5.4 Functions and namespaces

A namespace stores information about identifiers and the values to which they are bound. They are tables where Python looks up a name to find out which value or object it points to. There are three types of namespaces:

1. Local
2. Global
3. Built-in

A *local namespace* is created in each new block of code: Any identifiers created inside the block are local to that block and any blocks inside it. E.g. when you define a function, you create a new local namespace which holds the parameter names and any variables you introduce inside the function body. These names are local to the function, and once control returns from the function, its local namespace is deleted and its local variables are no longer accessible.

The *global namespace* contains whatever names you introduce in the main portion of your program on the outermost indentation level. These names are global in the sense that they are visible from inside functions as well as anywhere else in the program *after* the point where they were created. It also contains some other stuff including an identifier called `__name__` whose value is the name of the module in case the current code has been imported from another program (e.g., `math`), or the string `__main__` if the code is actually the main part of the program.

The *built-in namespace* contains all the identifiers Python "knows" a priori, i.e. before you start to create any yourself. You've seen things like `raw_input`,

`int` and `float`: These are functions Python are born with, and thus you can always use these names.

We should take a moment to go through in some detail what happens when you call a function. The small example program on the left in Figure 5.5 defines a function `f` which takes one argument called `a`. In the main program, a variable `z` is created and assigned the value 13, and then `f` is called on `z`; i.e., `f` is called with the argument `z`.

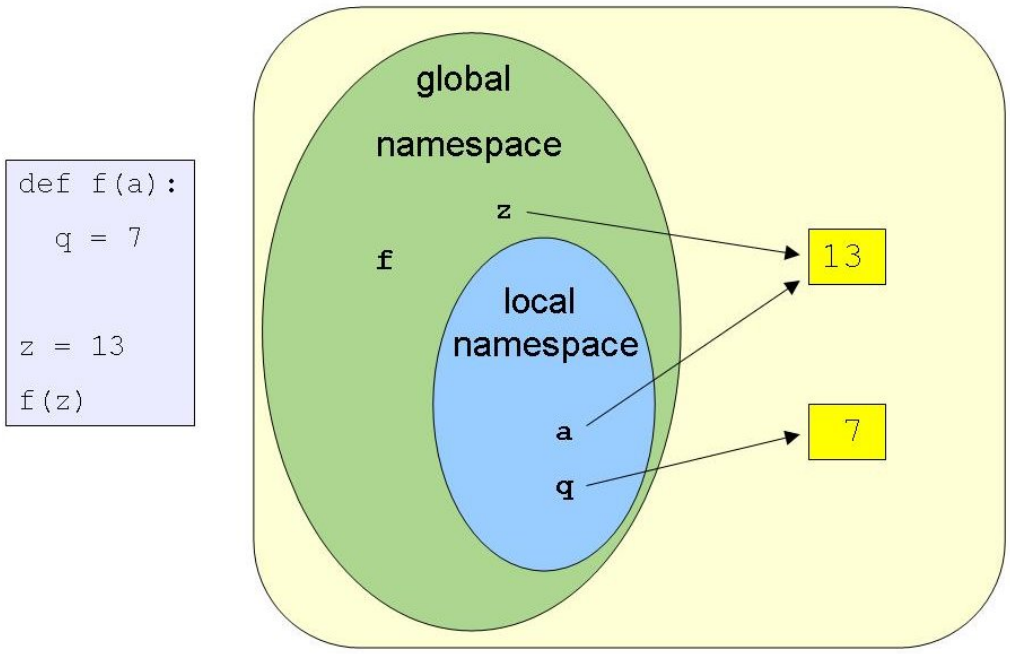


Figure 5.5: Calling a function.

When `f` is called on `z`, it doesn't mean that the identifier `z` itself is given to `f`. It means that the value of `z` is passed to `f`. Specifically, a *reference to the object in memory which* `z` *points to* is passed on to `f`. This reference is then assigned to a local variable `a`, created by the function, such that it points to the same object.

The process is illustrated in the right part of Figure 5.5. The overall memory is shown in bright yellow. The global namespace, shown in green, contains the identifiers `f` and `z`. `f` points to the function definition (not shown), while `z` points to the value 13, i.e. an object in memory containing the integer 13. The local namespace created by the particular call to `f` is shown in blue, and it contains two identifiers, `a` and `q`. `a` points to the *same object* which the function call's argument `z` does. `q` points to another memory location containing the integer 7. When the function call terminates, the local namespace is deleted. The 13-object continues to live on, however, since the global variable `z` still points to it. The 7-object is eventually removed from memory by the garbage collector.

Since `z` is not itself passed to the function, it continues to point to the 13-object during the course of the function call. Even if the function had modified `a`, only `a` would then be pointing to somewhere else in memory.

Numbers and strings are examples of *immutable* objects in Python. Immutable means "can't be changed". Thus, the specific object in memory holding the integer 13 is not changed if `a`, e.g., is incremented. Instead, a new object containing the integer 14 would be created.

The story is different when it comes to lists. Lists are *mutable*; i.e. objects representing lists *can* change. Imagine that our `z` was not pointing to an integer object but rather a list object. Then it would be possible to modify that same list from inside the function through `a`, since it references the same object that `z` does. It's probably not what you'd want, but it's possible. Caution needs to be taken when passing lists to functions. I'll return to that in Chapter 6.

When Python encounters some identifier in the program, it checks the three types of namespace in a specific order to find its value. First, local namespaces are checked, and if the identifier is found in a local namespace, its value is retrieved from there. If not, Python moves on to look up the identifier in the global namespace. If it is found, its value is retrieved from there. If not, finally the built-in namespace is searched and the identifier's value is retrieved. If the identifier is not found in any namespace, an error is reported.

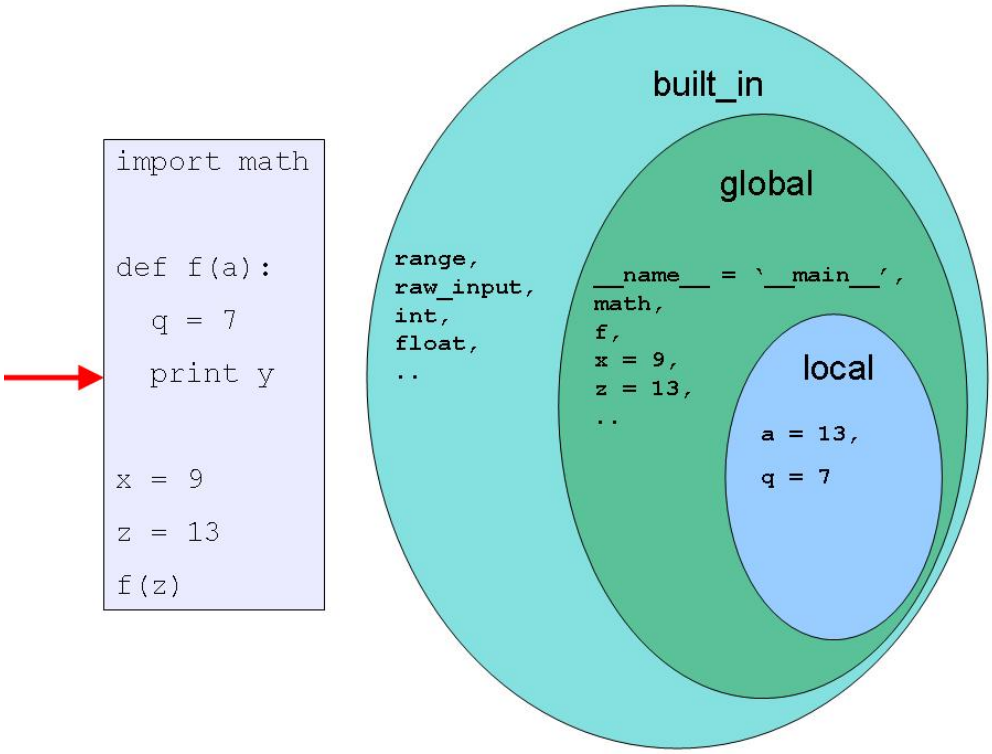


Figure 5.6: Namespaces: Searching for an identifier.

Look at the program on the left in Figure 5.6. It imports the `math` module, defines a function `f`, creates two variables `x` and `z`, and finally calls `f` on `z`. When the flow of control enters the function body and reaches the point marked by the red arrow, what happens?

The statement is `print y`. To fetch the value pointed to by `y`, Python first checks the local namespace. The local namespace consists of what is created inside the function `f`: The parameter `a` is local to the function; it holds (i.e., points to) the value 13 because a reference to this value was transferred to `f` by the function call `f(z)`. `q` is also local; it points to 7. There's no `y`, however, so Python moves on and checks the global namespace. Here it finds `__name__` which points to `"__main__"` (assuming the program is python'ed directly and not imported by some other program), the identifier `math` which points to the `math` module, `f`, the two variables created on the outermost level, `x` and `z`, and some other stuff. Still no `y`. Lastly, the built-in namespace is checked, but since `y` is not some inherent Pythonian identifier, it isn't found there either. Therefore the `print y` statement results in a `NameError: name 'y' is not defined`.

Figure 5.7 shows a slightly modified version of the same program. In the function `f`, the `print y` statement is gone, and instead of `q`, a variable `x` is assigned the value 7. In the main part of the program, a `print x` statement has been added. But which value does `x` have, 7 or 9?

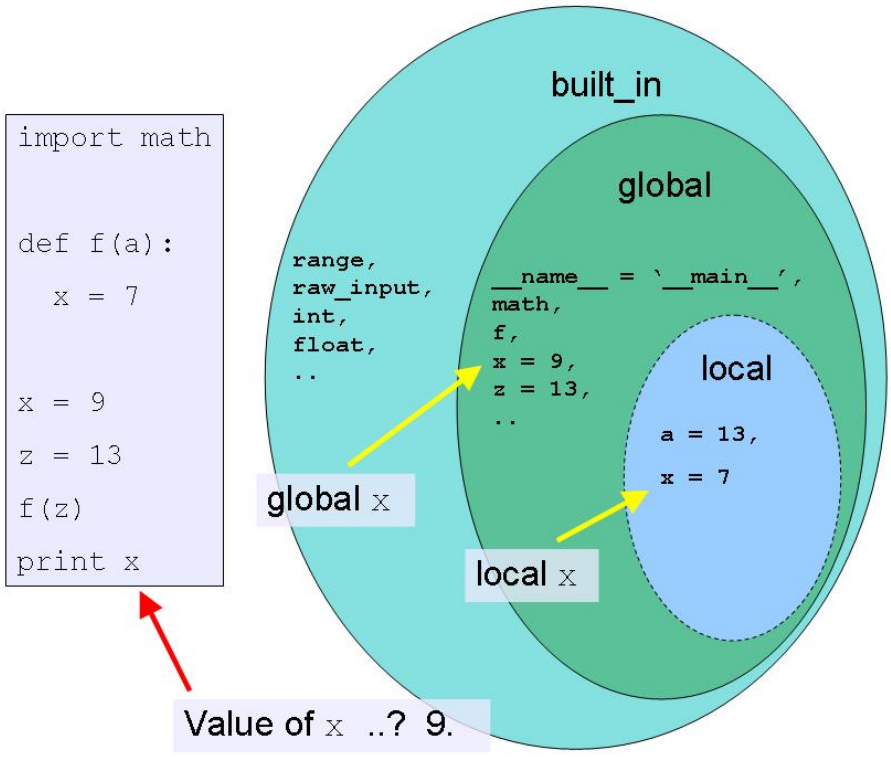


Figure 5.7: Namespaces: Identifiers with identical names.

Well, *inside* the function `f`, `x` would refer to the local variable created here. This `x` would shadow the one created on the outermost level: To fetch the value of an `x` inside the function, Python would start looking in the local namespace belonging to that function , and since it would find an `x` there, it wouldn't go on to consider other namespaces. The global `x` created on the outermost level lives on, though; it is just temporarily invisible.

But when Python reaches the `print` statement in the main program, marked by the red arrow, the function call's local namespace *no longer exists*: It is established at the beginning of the function call, but it vanishes when the call terminates. Thus, searching for an identifier called `x` to print, Python begins by looking in the global namespace since there are no current local namespaces, and here it finds an `x` pointing to 9, so a 9 is what's printed.

As explained, any identifier you create in the body of a function will be local, i.e. belong in the local namespace corresponding to the function. Let's be a little sophisticated: Imagine that you replace the line

```
x = 7
```

in the function `f` in Figure 5.7 with

```
x = x + 7
```

Now things become subtle: This line would create a local variable `x` shadowing the global `x` as before. But trying to assign to it the result of evaluating the expression `x + 7` forces Python to search for the value of `x`. Now it just created a local `x`, so it doesn't look further — but this `x` doesn't yet have a value! It is just set up as an identifier, but it doesn't point to something. This particular situation would actually give you an `UnboundLocalError: local variable 'x' referenced before assignment`, even though there *is* a global variable `x` holding a value.

If you need to modify a global variable from inside a code block, you need to tell Python. That's done with the keyword `global`. The program in Figure 5.8 is exactly the same as the one in Figure 5.7 except for the line `global x` in the function definition, shown in red. The difference is that after Python sees this statement, it will refer to the global namespace whenever it encounters an `x` and disregard the local namespace it would otherwise consult first. Thus, the line `x = 7` now *modifies the global* `x`, and hence the final `print` statement of the main program prints 7, not 9.

If there had been no `x` in the global namespace prior to the function call, it would have been created there; in other words, `x` would have survived the termination of the function call.

5.5 Scope

As mentioned, the scope of an identifier is the portion of the program where it is "visible", i.e. where it is accessible. This section deals with the question of

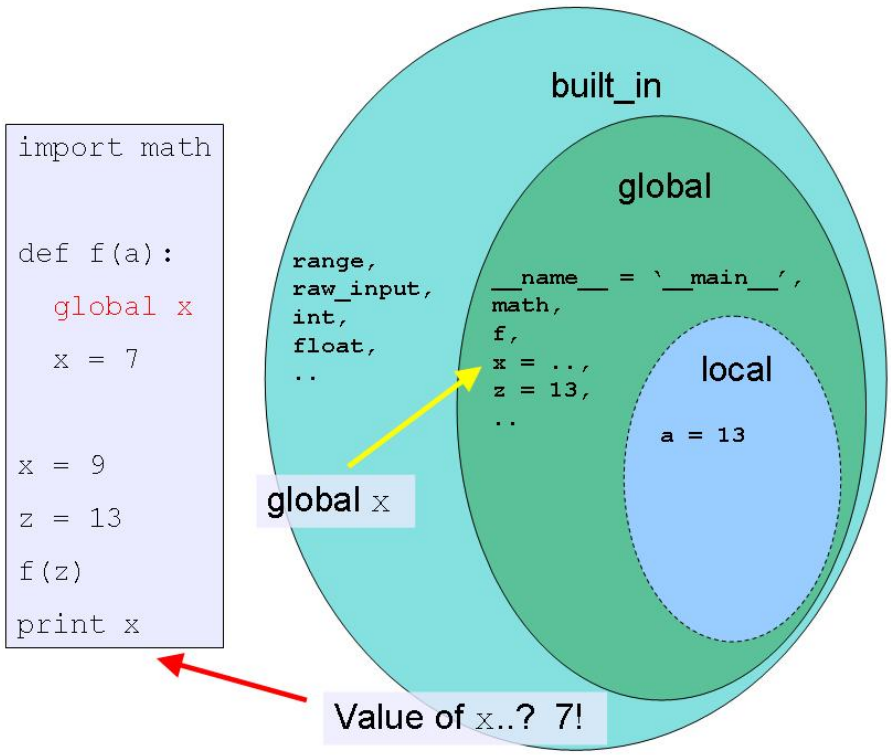


Figure 5.8: Namespaces: Modifying a global identifier inside a function.

where you can use the identifiers you create.

Local variables are created in a code block, and they are accessible from anywhere else in the same code block after they are created. Outside the code block, they're not. Global variables are accessible from anywhere in the program after they are created.

In both cases, "after they are created" means "in all code below their creation". Look at the examples in Figure 5.9 which illustrate two ways of conflicting with the scope rules.

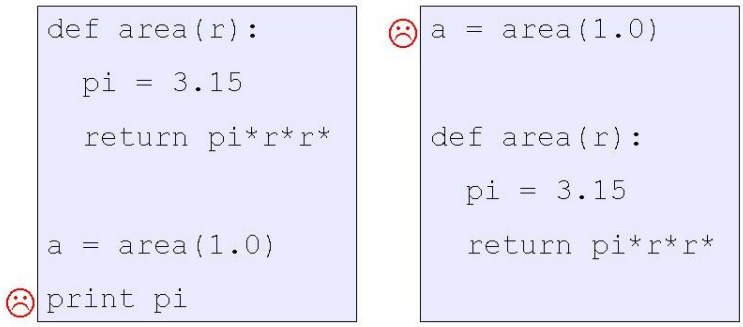


Figure 5.9: Using variables outside their scope.

In the left program, the variable `pi` is referenced outside its scope because it is a local variable belonging in the local namespace of the function call to `area`; it therefore doesn't exist anymore when the `print` statement is reached.

In the program on the right, a global function `area` is created — but the *call* to the function is attempted before the definition code is reached; the name is simply not known yet. Therefore, the call fails; the identifier `area` is used outside its scope.

A function is allowed to reference other functions defined *after* itself, though: When Python reads the program and reaches a function definition, the function body's code is not *executed*, and therefore a reference made to another function not yet defined does not cause an error; only when the function is *called*, Python requires references to actually point to something.

5.6 Default and keyword arguments

Sometimes a function is called repeatedly with the same arguments. A very convenient and elegant feature of Python is that you can give *default* arguments to your functions. They are set in the function definition following these rules:

- Default arguments must appear *to the right* of non-defaulted arguments:

```
def myfunction(a, b=2, c=3):
```

- Arguments passed in a call to the function are "filled in from the left":

```
myfunction(6, 3) # set a and b, use default for c
myfunction(6) # set a, use default for b and c
myfunction(6, 3, 7) # set all three, override defaults
```

The filling-in-from-the-left resolves any ambiguity which might otherwise arise. A priori it is not obvious what to do with two arguments if a function needs three. However, the function call `myfunction(6, 3)` is unambiguous since 6 and 3 are assigned from the left to the parameter names in the function definition, `a` and `b`, respectively. [1]

Another useful feature of Python is the possibility of using *keyword* arguments. Given a function with a very long parameter list, it surely becomes hard to remember the exact order in which the arguments should be passed. Imagine a function for transferring money between accounts:

```
>>> def transfer(sender, receiver, acc1, acc2, amount):
...     print "Transfer of $%d from account"%amount
...     print acc1
...     print "to account"
...     print acc2
```

[1] We'll meet Mr. Default Argument again in Section 6.7; Mr. Default Argument has.. issues, but before I can explain them, you need to understand a few other things. Meanwhile, don't worry, you won't notice.

```
...     print "Receiver:", receiver
...     print "Sender:", sender
...
>>> transfer("dad", "me", 123, 987, 100)
Transfer of $100 from account
123
to account
987
Receiver: me
Sender: dad
```

This function should correctly perform a transfer *from* the account `acc1` *to* the account `acc2`. In each function call, the name of the sender should be the first argument, the receiver the second, then the two account numbers should follow in the right order, and then finally the amount. If you, by unfortunate mistake, switch the two account numbers, you end up paying the money you should have received. This is not good.

What you can do is use the parameter names of the function definition as *keywords* to avoid any doubt as to what is what. Then the order of the arguments becomes irrelevant:

```
>>> transfer(receiver="me", amount=50, acc1=456,
acc2=987, sender="mom")
Transfer of $50 from account
456
to account
987
Receiver: me
Sender: mom
```

If you combine keyword and default arguments in the function definition, you don't have to explicitly name all parameters when you call the function. Again, defaulted arguments must appear to the right of any non-defaulted ones to avoid ambiguity, so we'll have to change the order of parameters in the function definition to default `me` and `acc2`:

```
>>> def transfer(acc1, sender, amount, receiver="me",
acc2=987):
```

This way, you can call the function (assuming the body is left unchanged) only using some of the required arguments and in the order you wish:

```
>>> transfer(acc1=789, amount=200, sender="grandpa")
Transfer of $200 from account
```

```
789
to account
987
Receiver: me
Sender: grandpa
```

You don't have to use keywords for all or none of the arguments in a function call, but if you do use keyword arguments, they must appear to the right of any non-keyword arguments in the call. It makes sense if you think about it; imagine a call `transfer(sender="grandpa", 200, 789)`: Is 200 the account number or the amount? If you try it, you'll get a `SyntaxError: non-keyword arg after keyword arg`.

The rule-of-thumb is that arguments in a function call which should be matched *by position* to parameters in the function definition must come first.

5.7 The `__name__` identifier

To illustrate a final point, let's revive the Caesar's Cipher program once more. You saw in Section 5.2 that importing the `Caesar_cipher2.py` file meant executing it, including the main program part. Back then, I suggested to simply take out everything but the functions so that the code could easily be imported without executing the "test program" also.

In some situations, however, it may be extremely useful if one could leave in such a test program. If one decides later to change the functionality of the module's real content, it would be nice to have a test program at hand which does nothing but test the functionality. That behavior is possible via the global identifier `__name__`: Recall that it holds either the string `"__main__"` or the name of the module being imported. That means you can use it to distinguish the two cases: If the program is being run as the main program, perform the test. If it is being imported as a module, don't perform the test.

Figure 5.10 shows a final version of the Caesar Cipher in which this check is implemented, and where the `shift` parameter in the function `caesar_cipher` is defaulted to 6.

```
1  def shift_char(c, shift):
2      if c.islower():
3          return chr(((ord(c)-97+shift)%26)+97)
4      elif c.isupper():
5          return chr(((ord(c)-65+shift)%26)+65)
6      else:
7          return c # don't shift non-letters
8
9  def caesar_cipher(s, shift = 6):
10     m = ""
11     for c in s:
12         m += shift_char(c, shift)
13     return m
14
15 if __name__ == "__main__":
16
17     # perform test:
18     import stringcolor
19     text = "Alia iacta est!"  # the die has been cast
20     cipher = stringcolor.inred(caesar_cipher(text))
21     print stringcolor.inred(text), "becomes", cipher
```

Figure 5.10: Program `caesar_cipher3.py`.

Now you know about:

- Character codes
- Built-in functions `chr` and `ord`
- Printing in red
- Importing modules
- The `randint` function from the `random` module
- How arguments are passed to functions
- Namespaces
- Keyword `global`
- Scope
- Default and keyword arguments
- A neat way of using the global identifier `__name__`

Chapter 6

Basic data structures

A *data structure*, sometimes called *data type*, can be thought of as a category of data. *Integer* is a data category which can only contain integers. *String* is a data category holding only strings. A data structure not only defines what elements it may contain, it also supports a set of *operations* on these elements, such as addition or multiplication. Strings and numbers are the core data structures in Python. In this chapter, you'll see a few more, almost as important, data structures. In the next chapter, you'll se how you can design your own, customary data structures.

The concept of a *sequence* is so fundamental to programming that I've had a very hard time avoiding it so far. And as you, alert and perky, have noticed, I actually haven't, since I involuntarily had to introduce sequences in Section 4.4 when talking about the `for` loop. In Python, the word "sequence" covers several phenomena. Strings are sequences of characters, and you heard about those in Chapter 3. In the coming sections, you'll hear about the two other basic types of sequence supported by Python: *Lists* and *tuples*. Later in this chapter, we'll get around to talking about *sets* and *dictionaries*.

Strings and integers represent concrete data objects; a string or a number represents true data in itself.[1] Lists, tuples and dictionaries are designed to *organize* other data, to impose structure upon it; they do not necessarily represent true data in their own right. For this reason, they are also called *abstract* data structures.

6.1 Lists and tuples — mutable vs. immutable

As I mentioned back in Section 4.4, lists and tuples are indexed just like strings: The first item of a sequence of length *l* has index 0, and the last has index *l-1*. You can get the length of a sequence with the built-in function `len`. You may use negative indices (e.g. -1 for the last item in the sequence), and you can slice out a (new) sequence using two or three parameters (e.g., `l[:3]` yields a new sequence containing the first three items of `l`). Go back and re-read Section 3.2 — everything in there applies to lists and tuples as well.

[1] — agreed? Let's not get lost in the obscure realms of philosophy.

Lists are written with square brackets `[1, 2, 3]` while tuples are written with parentheses `(1, 2, 3)`. When you *create* a list of values, you have to use square brackets with the comma-separated values inside; when you create a tuple, the parentheses are optional, and you may just write the values with commas in between. The commas create the tuple, not the parentheses — and square brackets create a list. See some examples, also refreshing the indexing business, below.

```
1  >>> l = ["George", "Paul", "John", "Pete", "Stuart"]
2  >>> l[0]
3  'George'
4  >>> l[3] = "Ringo"
5  >>> "Pete" in l
6  False
7  >>> del l[4]
8  >>> l
9  ['George', 'Paul', 'John', 'Ringo']
10 >>> t = ()
11 >>> t2 = ("A", "C", "G", "T")
12 >>> t3 = 7,
13 >>> t2[1:3]
14 ('C', 'G')
15 >>> t2[3] = "U"
16 Traceback (most recent call last):
17   File "<stdin>", line 1, in ?
18 TypeError: object does not support item assignment
```

In line 1, a list named `l` containing five strings is created. In line 4, the list's fourth element (index 3) is replaced with a new string. In line 5–6, it is checked that the former member element `Pete` is no longer present in the list. Using the `del` command, the list's fifth element (index 4) is deleted in line 7. This command can be used to delete any identifier as well. In line 8–9, the current content of `l` is printed.

In lines 10–12, three tuples are created: An empty tuple, a tuple with four string elements, and a so-called *singleton*, a tuple with only one element. Note that the comma in line 12 ensures that the variable `t3` is assigned a one-element tuple, not just the integer 7.

In line 13, a new tuple is created from a slice of `t2`, but when I attempt to change one of the members of `t2` in line 15, I get an error message: `TypeError: object does not support item assignment`.

This illustrates the one, very important aspect in which lists and tuples are functionally different: Lists are *mutable*, tuples are *immutable*. You may modify a list, but you can't modify tuples. More precisely: Once a list is created, you can replace or delete its elements, or even add new ones in any position. But once a tuple is created, it's take it or leave it. If you need to *pack together* some

items that logically belong together and never need to be replaced, you might use a tuple since that way not even erroneous modifications can be made. However, in all other cases, you should use lists. Both lists and tuples may contain data elements of any type, including compound data objects you have created yourself (we'll get to that in Chapter 7).

From 1960 to 1962, The Beatles had five members: John Lennon, Paul McCartney, George Harrison, Pete Best and Stuart Sutcliffe. In 1962, Sutcliffe left the band and Pete Best was replaced with Ringo Starr.

Adding elements to a list can be done using our old friend the + operator. Recall that that same operator is also used for adding up numbers and concatenating strings, but the overloading of this poor fellow doesn't stop here; you may also "add" two lists using +. Again, this makes perfect sense since the intuitive meaning of, e.g., `[1, 2, 3] + [4, 5]` is clear. You only have to keep in mind that if one of the operands of an addition operation is a list, then so must the other be. Thus, to add just a single element to a list, you have to put that single element in a list first before adding it: `[1, 2, 3] + [4]` works, but `[1, 2, 3] + 4` doesn't.

In fact, you can also multiply a list with a number using the well-known multiplication operator `*`. Here of course, at most one of the operands may be a list since it doesn't make sense to multiply a list with a list. Once more, the intuitive meaning is evident, and hence `*` is overladed to also work with a list:

```
>>> [0] * 7
[0, 0, 0, 0, 0, 0, 0]
>>> ['A', 'B'] * 2
['A', 'B', 'A', 'B']
```

Here we have to be a bit careful, though, and to understand why, we have to look into the computer's memory and see what actually goes on when you create a list. Figure 6.1 shows an example where a list with three elements has been created and assigned to a variable `a`. Recall that a variable assignment means that a reference (the horizontal arrow) is created associating some value (a chunk of memory, here the list) to a variable name. Each element in the list (marked by red bullets) *is also a reference* to some chunk of memory which, as mentioned, can represent any kind of data.

Now when you use the multiplication operator on a list, the list's elements are copied to form a new list. But the list elements are *references* to data, not actual data. Thus, no copying of data takes place. Imagine we execute the statement `b = a * 2` following the example of Figure 6.1. The impact on memory would be as shown in Figure 6.2.

The data elements, the yellow, green and blue blobs, are left unchanged and uncopied. The original list, `a`, is left unchanged. The new list, `b`, is a double copy of `a` in the sense that its first three members *reference the same objects* as do the elements of `a`, and so do the last three elements.

Now if the color-blob data objects are immutable, none of this causes problems. But if they are mutable, they may change. And any such changes would

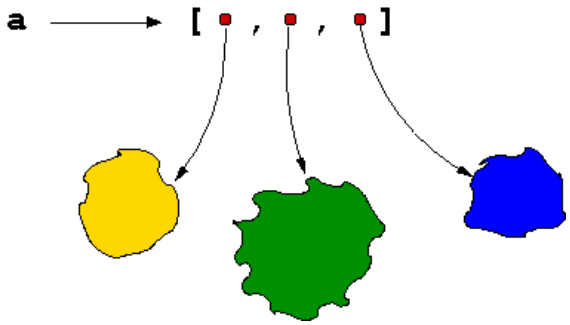


Figure 6.1: Creating a list.

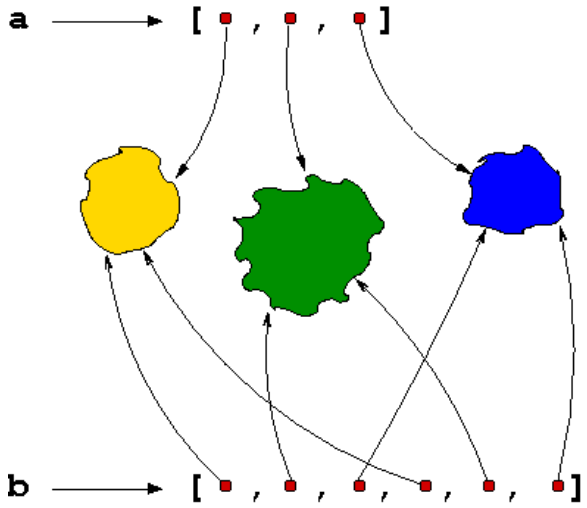


Figure 6.2: Result of executing the statement `b = a * 2` in the situation shown in Figure 6.1.

be visible through both `a` and `b`, since they reference the physically same objects. So far, you've heard mostly about immutable types of data: Numbers are immutable, strings are immutable, tuples are immutable. Thus, if the colored blobs are integers, floats, strings or tuples, they can't be modified and all is well. But we do know one mutable data type — lists ☺. And when we start dealing with classes and objects in Chapter 7, the same considerations apply.

To illustrate what may go wrong, here is a concrete example which corresponds precisely to Figures 6.1 and 6.2:

```
>>> a = [ ['Y'], ['G'], ['B'] ]
>>> b = a * 2
>>> a
[['Y'], ['G'], ['B']]
>>> b
[['Y'], ['G'], ['B'], ['Y'], ['G'], ['B']]
```

```
7  >>> a[0][0] = '**'
8  >>> a
9  [['**'], ['G'], ['B']]
10 >>> b
11 [['**'], ['G'], ['B'], ['**'], ['G'], ['B']]
```

In line 1, I create a list `a` containing three other lists (corresponding to the color-blob data objects of Figure 6.1), namely the one-element lists `['Y']`, `['G']` and `['B']`. Next, I create `b`. Now the situation is exactly as illustrated in Figure 6.2. Printing the contents of `a` and `b` in lines 3–6, one might think that the two are completely independent. However, the object pointed to by `a[0]` is a list (`['Y']`), a list is mutable, and so in line 7 I can change its first (and only) element `a[0][0]` to `'**'`. (Going back to Figure 6.2, you can imagine that the internal state of the yellow blob has changed. It's still there, all the references pointing to it are intact, but its state has changed). It is not surprising that the contents of `a` has changed, as shown in lines 8–9: What is disturbing is that `b` has also changed (lines 10–11) although I haven't touched it!

This is an example of a *side effect*. A side effect is when some programming action has (usually malign) effects other than those intended. In the above example, actually the problem arises from the internal one-element lists, not `a` and `b`. In general, whenever two variables point to the same, mutable object, care must be taken. And when using the multiplication operator on a list, what you get is exactly that: More references to the original data objects pointed to by the first list.

One further example. As mentioned (what, three times now?), lists may contain any kind of objects. If you need a 2-dimensional array, one way to go is to create a list containing other lists. For example, to represent the 3x3 identity matrix,

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

you might do like this:

```
I = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

This way, `I[0]` is the first row of the matrix, `I[1]` is the middle row, `I[2]` is the bottom row, and `I[0][2]` is the last element of the first row. If you want to initialize a 3x3 zero matrix (3 rows and 3 columns of 0's), you might consider using the multiplication operator:

```
1 >>> m = [ [0] * 3 ] * 3
2 >>> m
3 [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

```
4 >>> m[0][2] = 7
5 >>> m
6 [[0, 0, 7], [0, 0, 7], [0, 0, 7]]
```

The expression `[0] * 3` creates a list of three 0's. In general, the expression `[ X ] * 3` creates a list of three `X`'s, what ever `X` is. Thus, putting the two together in line 1 creates a list of three lists of three 0's. Print `m` (lines 2–3) and all looks well; at least it looks very similar to the identity matrix above. A lists of three lists, c'est bon, ça!

However, changing the last element of the first row to 7 (line 4) causes mayhem: Unfortunately, the last element of *every* row is changed. To see why, study Figure 6.3. When creating `I`, we explicitly created *three* new lists, one for each row (symbolized by the ugly magenta, light-blue and pink color blobs). When creating `m`, we first created *one* list of three 0's (`[0] * 3`, brown color blob), and then we created a list of three references to that same list.

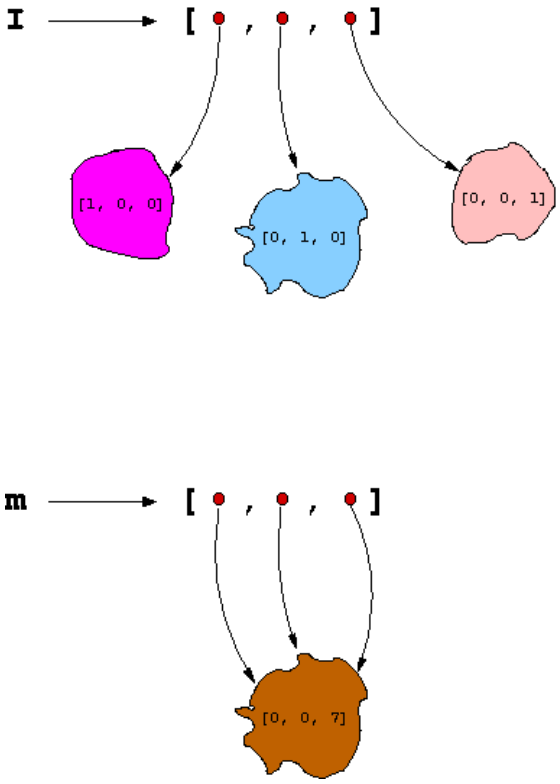


Figure 6.3: One right and one wrong way of creating a 3x3 matrix.

Thus, one way of inadvertently creating more references to the same mutable object is applying the multiplication operator to a list. Another very common one is through functions. As we saw in Section 5.4, when a function is called with some argument, actually a *reference* to that argument is what's passed to the function, not a copy of it. If this argument is a list (call it `arg`),

then again we have several references to the same mutable object. This means that any function taking a list as its argument can *modify* the original list.

Sometimes that's what you want — but sometimes it isn't. If not, you might consider using `arg[:]` for the list argument instead of just `arg`. Recall that `arg[:]` creates a slice (i.e., a new sequence) of `arg` which is actually the full sequence. In effect, this trick means shipping off a *copy* of `arg` to the function. There's always a "however", however: `arg[:]` creates a new list, but it doesn't create copies of the list's elements. Thus, the function can safely replace elements in its copy of the argument list without affecting the original list, but if the list's elements are mutable objects, the function can still modify them and thereby affect the original list. See the following demonstration and have a look at the cartoons in Figures 6.4 and 6.5.

```
1  >>> def modify_list(l):
2  ...   l[0] = 'modified'
3  ...
4  >>> def modify_listElement(l):
5  ...   l[0][0] = 'modified'
6  ...
7  >>> arg = [1, 2, 3, 4]
8  >>> modify_list(arg)
9  >>> arg
10 ['modified', 2, 3, 4]
11 >>>
12 >>> arg = [1, 2, 3, 4]
13 >>> modify_list(arg[:])
14 >>> arg
15 [1, 2, 3, 4]
16 >>>
17 >>> arg = [[1, 2], [3, 4]]
18 >>> modify_listElement(arg[:])
19 >>> arg
20 [['modified', 2], [3, 4]]
```

In lines 1–2, the function `modify_list` is defined. When called on a list, it replaces the first element of this list with the string `'modified'`.

Lines 4–5 define a second function, `modify_listElement`, which assumes its argument is a list of lists. When called, it replaces the first element list's first element with the string `'modified'`.

In line 7, a test list `arg` is created. Line 8 calls the first function on `arg` directly, and as lines 9–10 show, the original `arg` has been modified by `modify_list`.

In lines 12–13 we try again, create a test list `arg` and call the function. This time, however, we pass a copy of `arg` using `[:]`, and as lines 14–15 show, now the original `arg` is not modified by `modify_list`.

Finally, in line 17 `arg` is assigned a list of lists. When `modify_listElement`

function is called in the same manner as above on a copy of `arg`, it again modifies the original list, as shown in lines 19–20.

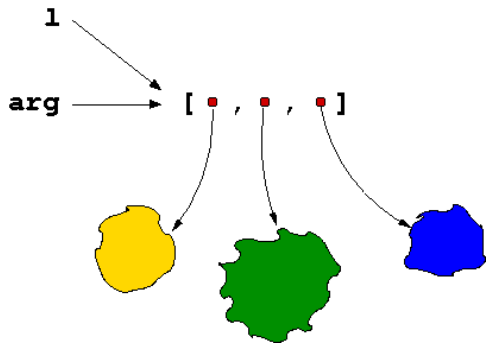


Figure 6.4: Passing an original list `arg` to a function `f(l)`. Any modifications made in `f` to its parameter `l` (e.g. replacing an element) affects the original argument `arg` as well.

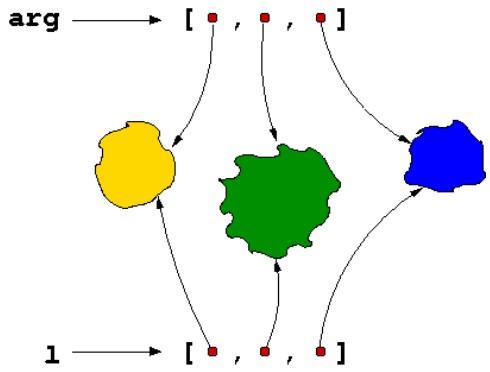


Figure 6.5: Passing a copy of an original list `arg` (e.g. with `arg[:]`) to a function `f(l)`. Any modifications made in `f` to its parameter `l` (e.g. replacing an element) do not affect the original argument `arg`; however, modifications made to the *elements* of `l` affect the original list as well.

Bottomline:

1. Lists are the first mutable objects you have encountered; take care if you've got several variables pointing to the same list.
2. Multiplying a list doesn't copy data, only references to data.
3. In a function call, the arguments passed on to the function are not copies of data, only references to data.

6.2 Creating and accessing lists

Rather than lists you explicitly type in element by element, mostly you'll be working with lists which are somehow automatically generated as the result of

some calculation or operation. As you've seen, several string methods return lists. Another very useful way to create sequences is via the built-in function `xrange`[2]. Since Chapter 3, you're familiar with the bracketed way of indicating optional arguments in functions, so here's the official, formal definition of `xrange`:

```
xrange([start,] stop[, step])
```

I know I don't have to explain the notation again, but nevertheless: `xrange` takes a mandatory `stop` argument and optional `start` and `step` arguments (..? Ah, good question! In case you give it two arguments, they'll be interpreted as `start` and `stop`, not `stop` and `step`. (Nice little poem there)). All arguments should be integers.

`xrange` produces a list of integers ranging from (and including) the `start` argument (if not given, 0 is default) up to but not including the `stop` argument. If `step` is given, only every `step`'th integer is included. Watch:

```
>>> xrange(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> xrange(7,10)
[7, 8, 9]
>>> xrange(0, 10, 3)
[0, 3, 6, 9]
>>> xrange(10, 0, -1)
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

You can't go backwards with positive steps, and you can't go forward with negative steps, so these commands produce empty lists:

```
>>> xrange(10,1)
[]
>>> xrange(0, 10, -3)
[]
```

It's time we reinvoke randomness. We've touched upon the `random` module in Section 5.3. From this module, you know the `randint` function which takes two arguments a and b and returns an integer N such that $a \leq N \leq b$. There is also a function called `randrange` which perhaps is a better choice: `randrange` returns a random integer from the list a call to `xrange` with the same arguments would generate. In other words: `randrange` and `xrange` have identical definitions (they both take a `stop` and optional `start` and

[2]There is also an equivalent function called `range`; however, `xrange` is more efficient for most purposes.

`step` parameters), and thus if you do `randrange(1, 5)`, you'll get either 1, 2, 3 or 4; *not* 5. This complies with the "up to but not including" convention used throughout Python when dealing with start and end indices, and so the definition of `randrange` is probably easier to remember than that of `randint`, where the end point is included in the list of optional values.

Let's check just how random the `random` module is. We'll do that by asking the user for a maximum number m. Then, over $100 * m$ rounds, we'll pick a random number between 0 and $m - 1$, and we'll keep track of which numbers were picked. On average, each number between 0 and $m - 1$ should be picked 100 times, yes? And conversely, if some number weren't picked at all, that would be suspicious, no? See the program in Figure 6.6.

```
1  from random import randrange

2  m = int(raw_input("Input maximum integer: "))
3  N = [0]*m

4  for i in xrange(100*m):
5      N[randrange(m)] += 1

6  notpicked = []
7  for i in N:
8      if i == 0:
9          notpicked += [i]

10 if notpicked:
11     print "These numbers were never picked: %s"%notpicked
12 else:
13     print "All numbers below %d were picked"%m
```

Figure 6.6: Program `randomness.py`.

In line 3, I create a list `N` of m 0's; `N[i]` will be the number of times i is picked. So far, no numbers have been picked, so `N` is initialized to all zeros. Recall that numbers are immutable, so there's no risk in creating a list of 0's this way.

In line 4, the loop starts; I want $100*m$ rounds, so I use a `for` loop to iterate through a list with precisely this length (generated with `xrange(100*m)`). In each round, I pick a random number between 0 and m with `randrange(m)`, and I use this number as the index into the list `N` and increment this entry by 1.

After the loop, I wish to collect all numbers which haven't been picked at all in a list called `notpicked`. So I go through all number counts in `N` (line 7), and if I see a count which is still 0, I add the index to `notpicked` in line 9 (using the generic shorthand notation `x+=y` for `x=x+y`). Note that in order to add the index `i` to the list, I first have to put in into a small list of its own with `[i]`.

Eventually I check whether any indices have been put into `notpicked`: As with the number 0, an empty list also is interpreted as `False` if used in a condition (non-empty lists are interpreted as `True`). So, if `notpicked` holds any values (line 10), print them. Otherwise, print the obligatory triumphant message. Figure 6.7 shows some test runs.

```
1  Input maximum integer: 10
2  All numbers below 10 were picked
3  Input maximum integer: 100
4  All numbers below 100 were picked
5  Input maximum integer: 1000
6  All numbers below 1000 were picked
7  Input maximum integer: 10000
8  All numbers below 10000 were picked
9  Input maximum integer: 100000
10 All numbers below 100000 were picked
11 Input maximum integer: 1000000
12 All numbers below 1000000 were picked
13 Input maximum integer: 10000000
14 All numbers below 10000000 were picked
```

Figure 6.7: Output from program `randomness.py`.

The last run with $m = 10,000,000$ took more than an hour on my computer, so I didn't bother going any further. Anyway, we don't detect any blatant flaws in Python's random number generation, although of course the statistical evidence is monumentally inconclusive in regard to the question about exactly how random the `random` module is. And perhaps you already know that in fact there is no such thing as genuine randomness in any computer[3]. What computers have are *pseudo-random* number generators. You'll most likely never know the difference since these number generators are very good, but it's sensible to keep in mind.

With a sequence of known length, you can *unpack* its contents into a set of named variables. Say that you know that some tuple `student` contains a student's first name, last name and ID number; then you can unpack it into three variables like this:

```
>>> student = ("Albert", "Einstein", "18977987")
>>> first, last, id = student
>>> id
'18977987'
```

[3].. and I'm not convinced that there exists true randomness in the real world either (give or take quantum mechanics). *Unpredictability* is what it is, but the distinction is purely academic.

This mechanism is actually very convenient. The program in Figure 6.8 demonstrates why. `L` is a list of early Nobel prize winners represented as tuples. Each tuple contains a name, a field, and a year. This is a typical application of tuples: Bundling together data that belongs together and that isn't necessarily of the same type.

```
1 L = [ ("Wilhelm Conrad Röntgen", "physics", 1901),
2       ("Ivan Pavlov", "medicine", 1904),
3       ("Henryk Sienkiewicz", "literature", 1905),
4       ("Theodore Roosevelt", "peace", 1906) ]

5 formatstring = "%-28s%-13s%s"
6 print formatstring%("Name", "Field", "Year")
7 print "-"*45
8 for name, field, year in L:
9     print formatstring%(name, field, year)
```

Figure 6.8: Program `unpacking.py`.

The `formatstring` defines the column widths of the table we're about to print out: First column should be left-centered and 28 characters wide; middle column should be 13 characters wide, and third column has no predefined length (it will hold the year of each Nobel prize award, and since no Nobel prizes were awarded prior to the year 1000, all the entries of this column will have four digits). The format string "expects" three arguments to be inserted, and in line 6 the table header line is printed by giving the column headers as arguments.

A line of suitable length is printed in line 7 (since strings are sequences, you are allowed to multiply them using `*`, as seen earlier). Finally, a `for` loop is used to iterate through all the elements in `L`. Typically, one would use a single variable as the counter (see below), but since we know that the entries are 3-tuples, we might as well use three distinct variables in the `for` loop, unpacking each tuple directly as we go. Figure 6.9 shows a run.

```
1 Name                         Field        Year
2 ---------------------------------------------
3 Wilhelm Conrad Röntgen       physics      1901
4 Ivan Pavlov                  medicine     1904
5 Henryk Sienkiewicz           literature   1905
6 Theodore Roosevelt           peace        1906
```

Figure 6.9: Output from program `unpacking.py`.

Unpacking can also be used as a quick way of swapping the values of two variables without the need for an explicit temporary variable. Rather than

```
temp = x
x = y
y = temp
```

you can simply do

```
x, y = y, x
```

Neat, eh? Technically, on the right-hand side of the = symbol, you create a tuple containing `y` and `x`, and this tuple is then unpacked into `x` and `y`, respectively.

6.3 List methods

As with strings, once you create a list you have access to a load of list methods that do useful things with it. Figure 6.10 shows them all, and below I'll explain some of them further. But before that, I'll say a few words more about some of the features of sequences that I have already touched upon.

The `del` command can be used to delete an element from a list, but it can also delete a *slice* of a list:

```
>>> L = [1, 2, 3, 4, 5]
>>> del L[1:3]
>>> L
[1, 4, 5]
```

It is also possible to *replace* a slice inside a list with some other sequence:

```
1  >>> L = [1, 2, 3, 4, 5]
2  >>> L[1:3] = (9, 9, 9, 9, 9)
3  >>> L
4  [1, 9, 9, 9, 9, 9, 4, 5]
5  >>> L[2:6] = "string"
6  >>> L
7  [1, 9, 's', 't', 'r', 'i', 'n', 'g', 4, 5]
```

Note that the slice in line 2, `L[1:3]`, has length 2 but the sequence whose values are inserted in its place have length 5. Note also that any sequence can be used in a slice replace, including strings which are sequences of characters; hence for strings, the individual characters are inserted in the list and not the string as a whole (see line 5). If you include a stepping parameter (extended slicing) for the slice to be replaced, the sequence to be inserted must have the same length, however:

Method name and **example on** `L=[1,7,3]`	**Description**

`L.append(item)`
 	
`>>> L.append(27)`
`>>> L`
`[1, 7, 3, 27]`
	Append `item` to `L`. `item` may be any kind of object. NB: Does not return a new list.

`L.extend(x)`
 	
`>>> L.extend([3, 4])`
`>>> L`
`[1, 7, 3, 3, 4]`
	Adds all the individual elements of the sequence `x` to `L`. NB: Does not return a new list.

`L.count(x)`
 	
`>>> [1, 2, 2, 7, 2, 5].count(2)`
`3`
	Return the number of occurrences of `x`.

`L.index(x[, start[, end]])`
 	
`>>> [1, 2, 2, 7, 2, 5].index(2)`
`1`
	Return smallest `i` such that `s[i]==x` and `start<=i<end`. I.e., look for `x` only within the start and stop indices, if given. Yields ValueError if x not in list.

`L.insert(i, x)`
 	
`>>> L.insert(1, 8)`
`>>> L`
`[1, 8, 7, 3]`
	Insert `x` at index `i`, keeping all the other elements. NB: Does not return a new list.

`L.pop([i])`
 	
`>>> L.pop()`
`3`
`>>> L`
`[1, 7]`
	Remove element at index `i`, if `i` is given. Otherwise remove last element. Return the removed element.

`L.remove(x)`
 	
`>>> L.remove(7)`
`>>> L`
`[1, 3]`
	Remove first occurrence of `x` from list. Yields ValueError if `x` is not in list. NB: Does not return a new list.

`L.reverse()`
 	
`>>> L.reverse()`
`>>> L`
`[3, 7, 1]`
	Reverse the list in-place (which is more space-efficient); i.e. no new list is returned.

`L.sort([cmp[, key[, reverse]]])`
 	
`>>> L.sort()`
`>>> L`
`[1, 3, 7]`
	Sort the list in-place (for space-efficiency). The optional parameters can specify how to perform the sorting (see text). NB: Does not return a new list.

Figure 6.10: List methods called on some list `L`.

```
>>> L = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> L[2:7:2] = "odd"
>>> L
[1, 2, 'o', 4, 'd', 6, 'd', 8, 9]
>>> L[2:7:2] = "some odd"
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: attempt to assign sequence of size 8 to
extended slice of size 3
```

You may also use `del` to delete an extended slice of a list.

Now on to the list methods shown in Figure 6.10; most are pretty straightforward, but still I'll briefly discuss some of them. First and foremost you should note that none of the methods return a new list; since lists are mutable, the list methods all perform their business directly on the given list without producing a new list (except for slicing).[4]

You've seen that one way of appending an element `x` to a list `L` is `L += [x]`. It's more elegant to use the `append` method and simply do `L.append(x)`; this way, you don't have to create a new singleton list first.

If you have two lists `a` and `b` and you want to concatenate them, you can use `a.extend(b)`. This adds all the elements of `b` to the end of `a`, keeping their order. And again: You don't get a new list after calling `a.extend`; the operation is performed directly on `a`. Thus, things like `c = a.extend(b)` are bad style: The list extension is actually performed, but the extended list is not assigned to `c` because `extend` doesn't return a new list.

The `index` method is obviously very useful. The optional parameters may also come in handy: Say that you have a long list of random integers, and that you want the indices of all 7's in the list. This small code snippet does the job (the `try/except` construct is just a way of catching the `ValueError` arising when no more 7's can be found; we'll get back to such *exception handling* later).

```
>>> a = [5,1,3,2,7,2,7,3,4,7,3]
>>> i = -1
>>> try:
...   while 1:
...     i = a.index(7, i+1)
...     print i,
... except:
...   pass
...
4 6 9
```

[4]With strings, the story is different: Strings are immutable, and therefore the generative string methods return new strings. Confer with Section 3.5.

The idea is to use an index pointer `i` as a pointer in the list and repeatedly search for the next 7 starting at index `i+1`. The result as calculated by `index(7, i+1)` is again stored in `i`, ensuring that the next search starts one position to the right of the 7 just found. Eventually, there are no more 7's, and we get the `ValueError`, but this error is caught by the `except` clause which simply executes a `pass` — a command which means "do nothing".

The `sort` method is efficient and versatile. It's also *stable*: The original order of two elements with the same sorting ranking is guaranteed not to change. With no given arguments, the list elements are just sorted in increasing order. Any list containing elements which can be pairwise compared with < and > can be sorted. If you want your list sorted according to some less than obvious criterion, it's possible to tell Python how by passing a *compare* function as an argument to `sort`. This function should take two arguments (list elements) and return -1 if the first should come before the second in the sorted list, 1 if it's the other way around, and 0 if you don't care (i.e. if the elements have the same rank according to your sorting criterion).

For example, look again at the list `L` of Nobel laureates from Figure 6.8. If we simply sort it with `L.sort()`, Python will sort its tuples according to the first element of each tuple. That would mean alphabetically by the prize winners' first name (and note again that `L.sort()` doesn't return a new list):

```
>>> L = [ ("Wilhelm Conrad Röntgen", "physics", 1901),
...        ("Ivan Pavlov", "medicine", 1904),
...        ("Henryk Sienkiewicz", "literature", 1905),
...        ("Theodore Roosevelt", "peace", 1906) ]
>>>
>>> L.sort()
>>> L
[('Henryk Sienkiewicz', 'literature', 1905),
('Ivan Pavlov', 'medicine', 1904),
('Theodore Roosevelt', 'peace', 1906),
('Wilhelm Conrad Röntgen', 'physics', 1901)]
```

Well, it just seems more Nobel prize-like to sort this esteemed group of people by their last name, and in order to do that, we write a special function which compares the last names of two laureates and pass this function to `sort`:

```
>>> def sort_laureate_tuples_by_last_name(a, b):
...   return cmp(a[0].split()[-1], b[0].split()[-1])
...
>>> L.sort(sort_laureate_tuples_by_last_name)
>>> L
[('Ivan Pavlov', 'medicine', 1904),
('Wilhelm Conrad Röntgen', 'physics', 1901),
```

```
('Theodore Roosevelt', 'peace', 1906),
('Henryk Sienkiewicz', 'literature', 1905)]
```

Ha-haa! The function receives two tuples `a` and `b`. First element of each (`a[0]` and `b[0]`) is a name. Using the string method `split` we divide each name string into separate words; taking the last of these (index -1) gives us the last name. These last names are then given to the built-in function `cmp` which compares its two arguments using < and > and returns -1, 1 or 0 depending on the relationship of the arguments, just the way we need it. Ideally, if the last names are the same, the function should compare the first names and base the ranking on that (since `sort` is stable, the pre-ordering by last name would not be destroyed by the following first-name sorting) — but I'll leave the details to you. When calling `sort`, you pass the name of your sorting function as an argument with no pair of parentheses after the name; parentheses would *call* the function rather than pass a reference to it. Just pass the function name to `sort` and it will call the function as it needs to.

The second optional parameter to `sort`, `key`, should be a function taking one argument and returning the value which will be subsequentely compared by `sort`. If you use this feature rather than a compare function, you should use `key` as a keyword argument (to prevent Python from attempting to use it as a compare function). See the following example where I sort a list of letters, ignoring the case:

```
>>> l = ['I', 'y', 'b', 'S', 'P', 'o']
>>> l.sort(key=str.lower)
>>> l
['b', 'I', 'o', 'P', 'S', 'y']
```

Here, I do not want to sort the characters the natural Python way; normally, an uppercase **X** would come before a lowercase **a** according to the Unicode character table (see Section 5.1). Thus using `sort` with no parameters would give me the sequence `I P S b o y`. To sort but ignore the case, I have to tell Python how, and in this case I do that by giving it a key function; this function will be called on each element (a character) and should return the key which the character should be ranked by. Here's an opportunity to use the `str` identifier I briefly mentioned at the end of Chapter 3! We need a function which takes one character (string) argument and returns the lowercase version of this string. We can't just use `lowercase`, because this identifier is not freely floating around; it's accessible either through a specific string or, like here, through the `str` thingy (which is actually a *class* — more on classes in the next chapter). So, each time `sort` needs to compare two characters of the list `l`, it uses the given key function, `str.lower`, to obtain the lowercase versions it should actually use for the comparison.

If you need to access a list method directly and not through a specific list, you can do it in exactly the same manner using `list`: `list.pop` is a function

which takes one argument, a list, and removes and returns the last element of this list. In general, when accessed through `list` and `str`, all the list and string methods take one extra parameter (expected to be the first in the parameter list), namely the list/string to perform the operation on.

As you see, you can easily pass a function as an argument to another function. Since all parameter passing is really through references, of course you can pass a reference to a function as well. Again, note the difference between `list.pop` and `list.pop()`: The first is a "passive" reference to a function, the latter *calls* the function. Finally, note also that strings are one kind of sequences, lists are another. Thus, you can't use list methods on strings and vice versa.

Due to their very mutability, lists are in many cases more efficient than strings in terms of memory usage. One typical example is when you're building a string little by little, e.g. by adding one character to it in every round of some loop. Each time you add a character to the working string, you'll get a new copy one character longer than the previous one. Say we want a random DNA sequence 100,000 nucleotides long. We might initialize an empty string and add a random nucleotide to it 100,000 times, as shown in Figure 6.11.

```
1 # bad way of building a string step by step
2 import random
3 nuc = ["A", "C", "G", "T"]
4 s = ""
5 for i in xrange(100000):
6     s += nuc[random.randrange(4)]
```

Figure 6.11: Program `buildstring.py`: An inefficient way of building a string step by step.

A random nucleotide is generated by using `randrange` to find a random index in the nucleotide list. This character is then added to `s` (line 6) — but since strings are immutable, in fact a *new* string is created and assigned to `s`. In the end, 99,999 strings have lived and died before the final string is complete. This takes up a lot of memory, although the garbage collector eventually frees this memory again, and it takes a lot of time to create all these strings.

```
1 import random
2 nuc = ["A", "C", "G", "T"]
3 l = []
4 for i in xrange(100000):
5     l.append(nuc[random.randrange(4)])
6 s = "".join(l)
```

Figure 6.12: Program `buildstring2.py`: It's better to build a list and convert it to a string.

Figure 6.12 solves the same problem in a much more efficient way. It uses a

list `l` instead, because since lists are mutable, the *same* list is extended with one nucleotide in each round of the loop. One list is all there ever is, and in the end (line 6), the individual characters of the list are joined (using the empty string as the delimiter) to form the complete DNA string. In other words: Whenever you need to gradually build a string, build a list instead and use the string method `join` in the end to convert the list into a string once and for all.

You may use lists and the list methods `pop` and `append` to build *stacks* and *queues*. A stack is a "last in, first out" data structure which contains data elements in a way such that when you add another element, you add it "on top" of the existing ones. When you remove one, you again pick it from the top. Thus, at any time you only deal with the top of the stack and have no access to elements further down. A stack can be implemented using a list and using `append(x)` to add an element `x` (to the end of the list), and `pop()` to remove an element (from the end of the list).

A queue is a "first in, first out" data structure, where new elements are inserted at the end, but where elements can only be removed from the front. You can implement a queue using a list and using `append(x)` to add an element `x` (to the end of the list), and using `pop(0)` to remove an element (from the front of the list).

6.4 Functions dealing with lists

Since sequences are fundamental to most programs, Python offers several built-in functions working with lists and tuples. As indicated, the typical scheme for accessing the elements of a sequence is the `for` loop, as in

```
for element in sequence:
  print element
```

You can do that for any sequence containing any kind of items: Numbers in a list, characters in a string, rows (lists) in a matrix (list of lists), etc. This way, `element` will in turn take the value of each item in `sequence`. You may also use several counter variables as shown in Figure 6.8 if your sequence contains other sequences.

Sometimes you not only want each item in a sequence but also the *index* of this item in the sequence. For such cases, the above method is insufficient as the index is not explicitly in play, and you're better off with this scheme:

```
for i, element in enumerate(sequence):
  print i, element
```

The expression `enumerate(sequence)` sets it up for you so that you can cycle through the sequence obtaining both the index `i` of each element and the `element` itself.

Next function on stage is `zip` which works as a kind of zipper. You supply a list of sequences, and it returns a list of tuples, where tuple i holds the i'th element of all argument sequences. It's best illustrated through an example (like everything):

```
>>> who = ['H. Sienkiewicz', 'I. Pavlov',
...        'T. Roosevelt', 'W. C. Röntgen']
>>> what = ['literature','medicine','peace','physics']
>>> when = [1905, 1904, 1906, 1901]
>>> for p, f, y in zip(who, what, when):
...   print "%s won the Nobel %s prize in %d"%(p, f, y)
...
H. Sienkiewicz won the Nobel literature prize in 1905
I. Pavlov won the Nobel medicine prize in 1904
T. Roosevelt won the Nobel peace prize in 1906
W. C. Röntgen won the Nobel physics prize in 1901
```

You can give it any number of input sequences (including strings which are then chopped up into characters); the resulting list of tuples will be truncated to have the same length as the shortest given argument sequence.

Another very useful function is `map`. This function requires a function and a sequence as arguments, and it then calls the given function on each element in the sequence and packs the results into a list. Thus, the i'th element of the returned list is the result of applying the given function to the the i'th element of the given sequence. Observe:

```
>>> def addVAT(a):
...   return a*1.25
...
>>> map(addVAT, [100, 220, 360])
[125.0, 275.0, 450.0]
```

The function `addVAT` adds a 25% VAT (value added tax) to the given amount. When this function is passed to `map` together with a list of amounts, the result is a list of the same amounts with added VAT.

If the function you want to map to all elements of some list takes more than one argument, `map` can still help you. You may supply more than one sequence after the function; in this case, the system is as with `zip`: The i'th element of the returned list is the result of the given function called on the i'th arguments of all the given sequences. Here's an example:

```
>>> import math
>>> map(math.pow, [2, 3, 4], [1, 2, 3])
[2.0, 9.0, 64.0]
```

The power function `math.pow` takes two arguments x and y and computes x^y. To use it with `map`, you therefore need to pass two sequences of

numbers as well. If we call these sequences X_i and Y_j, `map` will return the list $[{x_0}^{y_0}, {x_1}^{y_1}, \ldots]$. If the sequences don't have the same length, `None` is used for the missing values.

You also have a function called `reduce`. The idea behind it is to reduce all the elements of a sequence into one single element. The nature of the elements and the nature of the type of reduction is up to you, again through a *reduction* function which you supply. `reduce` needs two arguments: The reduction function (which should take two arguments and return one value), and a sequence. `reduce` works by calling the reduction function on the first two elements of the sequence, then on the result returned from the supplied function and the third element, etc., eventually returning one value. A starting value may be passed as an optional third argument — in that case, the first call to the reduction function will be on the starting value and the first element of the sequence.

One obvious example would be to use `reduce` to calculate the sum of all the numbers in a sequence of any length. You would build a function taking two numbers and returning the sum, and then you'd call `reduce` on this function and some list of numbers. `reduce` would then call your sum function on the first two numbers to calculate their sum; then on this sum and the third number to calculate this cumulated sum, etc. (Un)fortunately, however, this functionality already exists in Python through the `sum` function.

So, imagine instead that you are working with n-dimensional vectors. The Euclidian length of a vector (x_i) is by definition $\sqrt{\sum {x_i}^2}$. To calculate the length of any n-dimensional vector, we might use `reduce` the following way:

```
>>> import math
>>> def vector_length(v):
...     def calc(s, n):
...         return s + n*n
...
...     return math.sqrt(reduce(calc, v, 0))
...
>>> vector_length([3, 4])
5.0
>>> vector_length((2, 6, 3))
7.0
>>> vector_length((1, 2, 3, 4, 5, 3))
8.0
```

A helper function `calc` is defined inside the main function `vector_length`. That means that it is only accessible from inside the main function. This nesting of function definitions is not much used and not necessary either; its only claim to fame is that it signals to the world that the internal function is very closely associated to its mother function and only used by it and nowhere else.

`calc` takes two arguments, an "intermediate sum" and the next vector entry to be included in the sum, and it adds this entry squared to the sum and

returns the result. The main function `vector_length` calculates the Euclidian length of its argument vector `v` by calling `reduce` on `calc`, `v`, and the initial sum 0, and eventually taking the squareroot. As demonstrated, the function works on vectors of any length (and both lists and tuples). For the list `[2, 6, 3]`, `reduce` first makes the call `calc(0, 2)`. The result, 4, is used in the next call, `calc(4, 6)`, which yields 40. The final call, `calc(40, 3)` returns 49, and in the end, `reduce` returns 49 and the program goes on to return the square root of 49, namely 7.

`reduce` is a very general tool, and with a little imaginary effort, you can think of a variety of creative ways to use it — possibly even for useful things. The intermediate results passed down through the sequence don't necessarily have to be numbers; e.g., the simple, innocent-looking expression

```
reduce(str.join, ['do', 'you', 'get', 'this'])
```

yields this wonderful string:

```
tgydoodoueydoodouthgydoodoueydoodoutigydoodoueydoodouts
```

And now on to something completely different.

With the perhaps slightly more intuitive `filter` function you may filter out those elements from a sequence which do not qualify, according to a *filtering* function you provide which should take one argument and return either `True` or `False`. Only those elements that pass the filtering (i.e. for which the filtering function returns `True`), are put in the returned sequence. If you pass a string or a tuple, you'll get a string/tuple back; otherwise, you'll get a list. E.g., to create a new string with only the capital letters from some other string, you might call `filter`, passing the string method `str.isupper` as the work horse that actually does the filtering; `str.isupper` will be called on each individual character of the argument string, and only the uppercase letters are put in the result string:

```
>>> letter = """Sarah,
... my parents Erica and Elmer have convinced me that
... YOU are not my type of girl. And it
... is Time we don't See each other anymore.
... goodbye for EVEr,
... Norman"""
>>> filter(str.isupper, letter)
'SEEYOUATSEVEN'
```

Both `map`, `reduce` and `filter` are powered by a function you provide. Thus, their potential is bounded only by the helper function you design. You'll probably get the most benefit from using them if you need to do what they do more than once (or if the helper function you need already exists like in some

of the above examples); otherwise, you might usually do the same in one single `for` loop (although that won't earn you as much street credit, of course).

List comprehension is a concise way to create lists in a systematic manner. Let's jump right into an example. Earlier, I showed you how *not* to create a 3x3 matrix. Now I can provide you with a clever alternative. Brilliantly pedagogically, I'll first repeat the wrong way:

```
>>> m = [ [0, 0, 0] ] * 3   # don't do this
>>> m[0][0] = 77
>>> m
[[77, 0, 0], [77, 0, 0], [77, 0, 0]]
```

Only one `[0, 0, 0]` list is created which serves both as the upper, middle and bottom rows of the matrix. We need independent lists for the matrix' rows. List comprehension is the word! What we need is to create a new `[0, 0, 0]` three times and add them to a list. Observe:

```
1  >>> m = [ [0, 0, 0] for i in range(3) ]
2  >>> m[0][0] = 77
3  >>> m
4  [[77, 0, 0], [0, 0, 0], [0, 0, 0]]
5  >>>
6  >>> rows = 7; cols = 4
7  >>> m = [ [0] * cols for i in range(rows) ]
```

A list comprehension is written as a pair of brackets containing first an expression (representing each element of the resulting list) and then a `for` loop. In line 1, the "element expression" is `[0, 0, 0]` and the loop is `for i in range(3)`. The result is that in each round in the loop, a *new* list `[0, 0, 0]` is created and appended to `m`. In lines 7–8, a general expression for creating a *rows* x *cols* matrix is shown. You can even add an `if` statement after the `for` loop as a filter, demonstrated here:

```
>>> [ x*x for x in range(10) if x%3==0 or x%4==0]
[0, 9, 16, 36, 64, 81]
```

This code generates an ordered list where each element is a number of the form $x * x$; the x'es are taken from the list $0, 1, \ldots, 9$, but only those which can be divided evenly by 3 or 4 are used. Thus, the `for` loop (and any following `if`'s) generate a list, and from this list, the result list is created according to the element expression at the beginning. In the matrix example above, the element expression didn't reference the `for` loop's counting variable; here, it does.

Finally, there are also a range of functions that work on sequences of any length. `max`, `min`, `sum` calculate the maximum, minimum and sum, respectively, of all the numbers in the given list, regardless of its length. `max` and `min` even work for lists of strings, or just strings, too — as long as the elements in the given sequence can be compared using < and > (for strings and characters, this means a lexicographical comparison), they will work. There is also the function `sorted` which takes a sequence plus the same optional arguments as `list.sort` and returns a sorted *copy* of the given list. E.g., you might use it to loop over a sequence `s` in sorted order while keeping the original list unaltered:

```
for x in sorted(s):
```

6.5 Sets

Python has a data structure for *sets* in the mathematical sense: Unordered collections of items with no duplicates. Between two sets you can do union, intersection, difference and "symmetric difference" (yielding the elements from one but not both of the sets). You create a set from a sequence, and in the process, the sequence is copied and any duplicates are removed. Here are a few examples; first, let's create two sets:

```
1  >>> from random import randrange
2  >>> a = [ randrange(1, 21) for i in range(10) ]
3  >>> b = [ randrange(11, 31) for i in range(10) ]
4  >>> a
5  [4, 3, 8, 4, 15, 14, 14, 10, 19, 10]
6  >>> b
7  [14, 14, 17, 26, 28, 26, 18, 18, 13, 27]
8  >>>
9  >>> setA = set(a)
10 >>> setB = set(b)
11 >>> setA
12 set([3, 4, 8, 10, 14, 15, 19])
13 >>> setB
14 set([13, 14, 17, 18, 26, 27, 28])
```

The two lists `a` and `b` are generated using list comprehension: `a` contains 10 random numbers between 1 and 20 (both included), and `b` contains 10 random numbers between 10 and 30 (both included). From them I create the two sets `setA` and `setB` in lines 9 and 10, and when printing these sets in lines 11 and 13, I can check that the duplicates have indeed been removed. The sets appear to be ordered; in general, this is not guaranteed.

Next, let's perform some set operations. Line 1 below illustrates set difference (elements in `setA` but not in `setB`, i.e. 14 is out). Line 3 shows the union

of the two sets, i.e. all elements in the joint set with duplicates removed. Line 5 shows the set intersection, the elements in both sets — here only 14. And finally, line 7 shows the symmetric difference: The elements belonging to one but not both of the sets.

```
1 >>> setA - setB
2 set([3, 4, 8, 10, 15, 19])
3 >>> setA | setB
4 set([3, 4, 8, 10, 13, 14, 15, 17, 18, 19, 26, 27, 28])
5 >>> setA & setB
6 set([14])
7 >>> setA ^ setB
8 set([3, 4, 8, 10, 13, 15, 17, 18, 19, 26, 27, 28])
```

You can use sets in `for` loops instead of regular sequences; thus, a quick way of iterating through all elements of a list `L` ignoring duplicates would be `for x in set(L)`: The original list `L` would be untouched since the iteration would be performed on a separate set created from `L`.

6.6 Dictionaries

A dictionary in Python is an unordered collection of key/value pairs. The idea is that a dictionary maintains an association, also called a *mapping*, between each key and its corresponding value. Keys must be immutable objects (so that they may not be altered), and within the same dictionary, all the keys must be unique. In the good old days, the obvious simple, introductory example was to create a dictionary representing a phone book. You know, you look up a person and get the person's phone number. In the dictionary, the names would be the keys, and the numbers would be the values. However, these days any person over the age of 5 apparantly has at least 3 phone numbers, so the same direct, one-to-one correspondence has not survived progress; as explained, you can't insert the same name (key) twice with two different phone numbers (values) in a dictionary. So either, we should find another example, or each value should be a *list* of phone numbers instead. I choose A).

In Denmark, everyone has a unique CPR identity number (a Central Person Registry number) composed by the person's birth date, a dash, and a four-digit code. You're given this number moments after you are born. Thus, we can think of the CPR as a giant dictionary where CPR numbers are the keys, and people (in our simple case, names) are the values. We'll use strings for the CPR numbers since they contain a "-". There are basically two ways of initializing a dictionary, both demonstrated in the following example.

```
>>> cpr = { '130371-3121':"Anders Andersen",
...         '050505-0310':"Trylle Espersen",
```

```
...          '121123-2626':"Ib Ædeltville-Hegn"}
>>>
>>> cpr = dict([('130371-3121', "Anders Andersen"),
...             ('050505-0310', "Trylle Espersen"),
...             ('121123-2626', "Ib Ædeltville-Hegn")])
```

The first way is using curly braces. You can create an empty dictionary with just `cpr = {}`, or you can put content in it by listing a comma-separated set of `key:value` pairs. The other way is using the built-in method `dict` (which is similar to `str` and `list`); you call it with a list of tuples, each tuple containing a key and its associated value.

You can access (look up) the value of some key the same way you index a list; instead of an index number, you use the key:

```
>>> cpr['050505-0310']
'Trylle Espersen'
```

Attempting to look up the value for a non-existant key is an error, as demonstrated in lines 1–4 below. You may use the same notation to replace an existing value for some key (line 6), and also to add a new key/value pair to the dictionary (line 7). This last feature constitutes a potential pitfall: If you wish to replace the value of some key but then misspell it, you won't get a warning; instead, you will have added a new key/value pair while keeping the one you wanted to change. To remove an item from a dictionary, use `del` (line 8).

```
1  >>> cpr['290872-1200']
2  Traceback (most recent call last):
3    File "<stdin>", line 1, in ?
4  KeyError: '290872-1200'
5  >>>
6  >>> cpr['130371-3121'] = "Anders Ivanhoe Andersen"
7  >>> cpr['250444-7411'] = "Niels Süsslein"
8  >>> del cpr['050505-0310']
9  >>>
10 >>> cpr
11 {'130371-3121': 'Anders Ivanhoe Andersen',
12 '250444-7411': 'Niels Süsslein',
13 '121123-2626': 'Johan Ædeltville-Hegn'}
```

Printing the contents of the dictionary `cpr` in lines 10–13 demonstrates that the items of a dictionary do not appear in any order, just like in sets. The most recently added item appears in the middle of the printed list.

As with strings and lists, a dictionary is pre-equipped with a bunch of ready-made dictionary methods listed in Figure 6.13. I think from the explanation in the figure, most of them are straightforward; below I'll briefly go over a few.

Method name	**Description**
`D.clear()`	Delete all items from `D`.
`D.copy()`	Return *shallow* copy of `D`; i.e. all items in the copy are references to the items in `D`.
`D.get(key [, defaultValue])`	Return the value associated with `key`. If `key` is not in the dictionary, return `None` or the optional `defaultValue` if it is given.
`D.has_key(key)`	Return `True` if `D` contains the given key, `False` otherwise.
`D.items()`	Return the key/value pairs of `D` as a list of tuples.
`D.keys()`	Return a list of all keys of `D`.
`D.pop(key [, defValue])`	Remove `key` from `D` and return its value. If `key` is not found in `D`, return `defValue` if it is given, otherwise raise KeyError.
`D.popitem()`	Return an arbitrary key/value pair of `D` as a tuple.
`D.setdefault(key [, defValue])`	If `D` doesn't already contain `key`, add it with value `None` or `defValue` if it is given. If `key` is already in `D`, do nothing. In all cases, return the value of `D[key]`.
`D.update(D2)`	Add all items of the dictionary `D2` to `D`, overriding the values of those keys in `D` that also exist in `D2`.
`D.values()`	Return a list of all the values in `D` (i.e., not the key/value pairs, just the values).
`D.iterkeys()`	Return an iterator of all the keys of `D`.
`D.iteritems()`	Return an iterator of all the key/value pairs of `D`.
`D.itervalues()`	Return an iterator of all the values of `D`.

Figure 6.13: Dictionary methods called on some dictionary `D`.

The difference between a *shallow* copy and a *deep* copy is something we have already explored to painful depths with lists. If `d` is a dictionary, then `d.copy()` returns a dictionary with the same keys as `d` which all reference the same objects as the keys of `d`. This becomes a problem if some of the values are mutable objects: Modifying a mutable value through `d` would then affect the copied dictionary as well. The same applies to the `update` method: The keys added to `d` through a `d.update(d2)` call point to the same objects as do the keys of the second dictionary `d2`.

Three methods return a so-called *iterator* object. For now, all you need to know about iterators is that they are designed to iterate efficiently through the elements of a sequence. Thus, you can use an iterator in any `for` loop.

In general, there are also several ways to iterate through all the items in a dictionary. If you need both key and value for each item inside the loop, you might use the `iteritems` methods:

```
for c, name in cpr.iteritems():
```

You could also do

```
for c, name in cpr.items():
```

but the difference is that with the latter option, the entire set of dictionary items is copied into a new list (of *references* to the items, that is, but still). With the first option (and with iterators in general), items are not copied before they are accessed, and they are accessed one at a time as the user needs them. For dictionaries, it should hardly ever be a problem, but when we get to discussing reading files, the same principle forcefully applies.

If you just need the keys, you should use this syntax:

```
for key in cpr:
```

instead of the equally legal

```
for key in cpr.keys():
```

The reason is the same: Calling the method `keys` copies the entire set of keys to a new list. The `for key in cpr:` way implicitly retrieves an iterator from the dictionary and uses that to access the keys one by one, avoiding the copying. In fact, this syntax is equivalent to `for key in cpr.iterkeys()`, one of the other methods listed in Figure 6.13.

One other dictionary method which you would use directly through `dict` deserves mention. It is defined as `fromkeys(S [, defaultValue])`, and you can use it to initialize a dictionary with keys taken from a given sequence `S`. All keys are assigned the optional `defaultValue`, or `None` if it isn't given. You may use any sequence; each element of the sequence will become a key in the new dictionary (see lines 1–2 below).

```
>>> dict.fromkeys('abcde')
{'a': None, 'c': None, 'b': None, 'e': None, 'd': None}
>>>
>>> d = dict.fromkeys(xrange(1, 6), [])
>>> d[1].append('NB')
>>> d
{1: ['NB'], 2: ['NB'], 3: ['NB'], 4: ['NB'], 5: ['NB']}
```

The one initialization value is really just that one same value used for all keys. I.e., when using a mutable object, once again you have to beware as demonstrated in lines 4–7. Giving an empty list as the default value should ring the alarm bell in the back of your head, and indeed it is *not* a new, fresh, empty list that is assigned to each key: Appending the string `'NB'` to the list pointed to by the key `1` affects all the list of all the other keys as well.

The previous section introduced sets. Python implements sets using dictionaries: Set members are unique, and so are dictionary keys, so the elements of a set are actually represented as the keys of a dictionary (associated with some dummy value). This has the implication that you can't create a set of items which would not be legal dictionary keys. E.g., lists can't serve as dictionary keys because they are mutable, and hence they can't be put in sets either.

As a final remark in this section, let me introduce to you a way of actually performing a "real" copy of some object. All this talk about several references to the same object may be a little puzzling, especially so since there are no problems as long as the object is immutable. If you really need to copy a dictionary or a list such that all the referenced data, and not just each reference, is in fact copied, there's always the `deepcopy` function of the `copy` module. Observe:

```
>>> from copy import deepcopy
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = [[7,8], [9, 10]]
>>> d = {1:a, 2:b, 3:c}
>>> d
{1: [1, 2, 3], 2: [4, 5, 6], 3: [[7, 8], [9, 10]]}
>>>
>>> d2 = deepcopy(d)
>>> d2[2] = 'NB'
>>> d2[3][0][1] = 'NB'
>>> d
{1: [1, 2, 3], 2: [4, 5, 6], 3: [[7, 8], [9, 10]]}
>>> d2
{1: [1, 2, 3], 2: 'NB', 3: [[7, 'NB'], [9, 10]]}
```

First I define three lists `a`, `b` and `c`, the latter of which is a list of lists. Then I create the dictionary `d` with the keys 1, 2 and 3, each pointing to one of the lists. In line 9, I create a *deep copy* `d2` of `d`, i.e. a copy where *all* data pointed to by `d` is copied. The "deep" refers to the copying mechanism which follows all references recursively, i.e. following references to references to references, copying every data object it meets, until it reaches the bottom (or something it has already copied). This is proven by the fact that I can change the second element of the first list in the list pointed to by key 3 (see lines 11 and 15) in the new dictionary `d2` without touching the original `d`.

Just as a small exercise, what would happen if I deleted the last element from the list `a`..? Anyone?

```
1  >>> del a[-1]
2  >>> d
3  {1: [1, 2], 2: [4, 5, 6], 3: [[7, 8], [9, 10]]}
4  >>> d2
5  {1: [1, 2, 3], 2: 'NB', 3: [[7, 'NB'], [9, 10]]}
```

Since `d`'s key 1 references `a` (i.e. the same list that `a` references!), `d` is modified when I modify `a`. However, being a deep copy, `d2` is not.

6.7 An issue concerning default arguments

As promised, I now return briefly to the concept of default arguments, introduced in Section 5.6. If one uses a mutable value as the default argument of a function, one might be in trouble. Figure 6.14 has the story.

This program uses a dictionary `d` to record a selection of nations and the number of times they have won the soccer World Cup. The dictionary is initialized in line 7, and in lines 8–11 I call the function `init_team` on `d`, a nation's name, and a list of years this nation has won the Cup — except that for Sweden and Denmark, I don't pass any list since neither Sweden nor Denmark have won the Cup.

Turning to `init_team`, we indeed see that it takes three arguments; a dictionary, a team, and a list which is defaulted to the empty list. The function simply inserts a new entry in the dictionary with the team name being the key and the list being the value.

In lines 12 and 13, I call another function, `win`, to update the dictionary with two more World Cup winners. This function takes a dictionary, a team and a year and simply appends the year to the entry in the dictionary corresponding to the team. Finally, in lines 14–16 I print out the contents of the updated dictionary.

Figure 6.15 shows the output when running the program. Lines 1–6 look okay; they testify to the calls to `init_team` and `win`. When I print the dictionary, it doesn't look okay, though: Apparantly, the Swedes claim that they win the 2010 Cup even though I added this year to Denmark's list!

```
def init_team(d, team, winlist=[]):
    print "Adding %s to the table"%team
    d[team] = winlist

def win(d, team, year):
    print "%s wins the world cup in %d"%(team, year)
    d[team].append(year)

d = {}
init_team(d, "Brazil", [1958, 1962, 1970, 1994, 2002] )
init_team(d, "Italy", [1934, 1938, 1982] )
init_team(d, "Sweden")
init_team(d, "Denmark")
win(d, "Italy", 2006)
win(d, "Denmark", 2010)
print
for team, winlist in d.iteritems():
    print team, winlist
```

Figure 6.14: Bad, bad program `worldcup.py`!

```
Adding Brazil to the table
Adding Italy to the table
Adding Sweden to the table
Adding Denmark to the table
Italy wins the world cup in 2006
Denmark wins the world cup in 2010

Brazil [1958, 1962, 1970, 1994, 2002]
Denmark [2010]
Sweden [2010]
Italy [1934, 1938, 1982, 2006]
```

Figure 6.15: Output from program `worldcup.py`.

The explanation again has to do with the mutability of lists. Once Python reads the definition of the `init_team` function and sees that the parameter `winlist` has a default value, it creates this default value, an empty list, and sets up `winlist` to point to it. When the function is called for the first time with no argument passed to `winlist` (in our exampe when initializing Sweden), the empty list pointed to by default by `winlist` is used as the value of Sweden's dictionary item.

But — the local identifier `winlist` lives on in the function's namespace. Thus, after the function terminates, `winlist` still points to the list now used

by Sweden. In further calls to `init_team` passing no argument to `winlist`, like the one initializing Denmark, the *same* default list as before is used in the new dictionary items. Sweden's list! Thus, when inserting the year 2010 in Denmark's list, the Swedes see this update and rejoice too. Major bummer!

We learn from this example that a default argument is not assigned anew every time the function is called. It is assigned *once and for all*, and if the default argument is mutable and you modify it — or if you re-assign the defaulted parameter to some other value — the changes will persist in future calls to the function.

Instead of using lists as default arguments, use this work-around:

```
def init_team(d, team, winlist=None):
    if not winlist:
        winlist = []
    print "Adding %s to the table"%team
    d[team] = winlist
```

This way, a *new* empty list is created each time the function is called with no argument passed to `winlist`. I think now you have received decent schooling in the world of references, yes?

6.8 Not-so-small program: Premier League simulation

As this information packed chapter is drawing to a close, I'll throw in a larger example bringing together all the things you've experienced in the last 30 pages.

I like soccer, and back in the eighties, in the heyday of the Commodore 64, I spent a fair amount of time writing simulations of soccer tournaments. Such a simulation might go like this: From a given list of teams, set up a tournament where all teams play each other and the outcome of each match is somehow determined by randomness. Keep track of all statistics (points, goals scored, etc. The works). Output an ordered ranking list in end.

Here, I'll simulate a small version of England's Premier League, and I'll give each team a *strength* such that the program may capture the fact that some teams are in fact better than others. So why not jump right into it? I've split the program in two files, `premierleague.py` and `premierleague_fncs.py`. The first is the main program, the second contains some functions used by the main program. In the following figures, I'll show you the various parts of the files as I discuss them.

Figure 6.16 shows the first part of the main program. I import everything from the functions module in line 1, and next I define the teams of the tournament as a list of tuples and assign it to the variable `teams`. Each tuple contains a team name and a strength (a number between 1 and 10). I'm going to be using 3-letter versions of the full team names as team IDs, so in line 15 I call a function designed to return a list of such short names from the given list of

full names. This function is explained below. The list of IDs is assigned to the variable `ids`.

```
1  from premierleague_fncs import *

2  # list of (team, strength) tuples:
3  teams = [("Manchester United", 10),
4           ("Chelsea", 10),
5           ("Bolton", 7),
6           ("Arsenal", 9),
7           ("Everton", 7),
8           ("Aston Villa", 6),
9           ("Manchester City", 5),
10          ("Liverpool", 8),
11          ("Tottenham", 6),
12          ("Wigan", 4)]

13 # build list of 3-letter versions of team names to be
14 # used as unique IDs:
15 ids = createShortNames(teams)

16 # build a dictionary where each key is a short name (ID)
17 # and the corresponding value is a
18 # (full team name, strength) tuple:
19 id2team = dict( zip(ids, teams) )

20 # build a dictionary representing the match board:
21 matchboard = initializeMatchBoard(ids)

22 # build a dictionary such that each team ID is
23 # associated with a tuple of 7 integers representing
24 # played matches, matches won, matches drawn,
25 # matches lost, goals scored, goals received, and points
26 stats = dict([(id, [0]*7) for id in ids])
```

Figure 6.16: Program `premierleague.py`, part 1.

To be able to efficiently look up a team's full name from its short ID, I create the dictionary `id2team` in line 19. I use the `dict(...)` way of initializing it; to do that, `dict` needs a list of (key, value) tuples, and that's exactly what I get from using the function `zip` on the two lists of IDs and teams. `zip` zips together the first ID of the `ids` list with the first team tuple of the `teams` list, and this pair then becomes a key and value, respectively, of the dictionary.

To remember all match results, I'll define a *match board*. It's supposed to simulate a big table with team names on both the x- and y-axis; as the tournament progresses, the result of each game is inserted in the table entry corresponding to the two teams that played the game. Such a match board is created by a function in line 21. I'll explain how in just a minute.

For each team I wish to keep track of the number of games played, games

won, games drawn and games lost; also, the number of goals scored and goals received; and, of course the total number of points earned (3 for a win, 1 for a draw, 0 for a loss). I.e., in total, I want to keep seven values updated for each team during the tournament. Again I'll use a dictionary to look up these statistics for each team using its short name ID as key. The statitics themselves I will represent as a list of integers, initialized to seven 0's.

This bit is carried out in line 26 of Figure 6.16. Again, `dict` needs a list of tuples to initialize a dictionary, and this time I use list comprehension to create it. The expression `[(id, [0]*7) for id in ids]` creates a list of pairs where the first item of each pair is an ID from the `ids` list and the second item is a list of seven 0's. Remember that the list comprehension machinery creates a *new* list of 0's for each id.

Figure 6.17 shows the first part of the file `premierleague_fncs.py`, including the two functions we've used already in `premierleague.py`. Let's go over `initializeMatchBoard` first; its job was to initialize a big table for holding the results of all matches. I'll record the matches using a dictionary where the keys are tuples of the form (ID_1, ID_2). Tuples are immutable, so I'm allowed to use tuples as dictionary keys. The value associated with, e.g., the key `(Eve, Wig)` should eventually be the string `'5-0'` if Everton wins their home game against Wigan $5 - 0$. For the initialization, I'll use an empty string of three spaces (for pretty-printing purposes to be unveiled later). Note that all teams will meet each other both at home and away, so there'll be an `(Eve, Wig)` key as well as a `(Wig, Eve)` key.

To create this dictionary, I again use a list comprehension. The expression `[(a, b) for a in ids for b in ids]` creates a list of all combinations of tuples `(a, b)` where `a` and `b` are from the list of IDs, `ids`. This list is generated in line 35 of Figure 6.17 and assigned a local variable. Next, in line 36 this list is given to the dictionary method `fromkeys` which besides a list of keys also may take an initialization value to assign to all the keys. This is where I give the 3-space string. The dictionary thus created by `fromkeys` is then returned by the function `initializeMatchBoard`.

The job of the `createShortNames` function of Figure 6.17 is to semi-intelligently create a unique three-letter ID string for each of the team names it is given in the parameter list `teams`. As it is supposed to return a list of these IDs, I start by initializing an empty list, `shortnames`, in line 5. In line 6, the loop through the `teams` list begins. Since `teams` is in fact a list of (team name, strength) tuples, I can unpack each list item into two variables `team` and `strength`. Next, in line 7 I use the string method `split` to split the team name into separate words (by default splitting by each whitespace character). If the name contains more than one word (`len(words)>1`, line 8), I want the ID to end with the first letter of the second word. Thus, in line 9 I create a string `s` containing the first two letters of the first word of the team name, `words[0][:2]`, plus the first letter of the second word, `words[1][0]`. Note that in the slicing operation, I don't provide the starting index because it defaults to 0. If there is only one word in the team name, I simply use the first three letters as the short name (line 11).

```
from random import randrange, gauss

def createShortNames(teams):
    """Return new list of unique, 3-letter
    abbreviations of all the names in the given list"""
    shortnames = []
    for team, strength in teams:
        words = team.split()
        if len(words) > 1:
            s = words[0][:2] + words[1][0]
        else:
            s = words[0][:3]
        while s in shortnames:
            s = raw_input("""Enter 3-letter name
            for %s not in %s"""%(team, shortnames))
        shortnames.append(s)
    return shortnames


def longestName(length, teamtuple):
    """Return maximum of given length and given team's
    name, where team is a (team name, strength) tuple"""
    return max(length, len(teamtuple[0]))


def initializeMatchBoard(ids):
    """Return dictionary representing an initialized
    match board holding all results of all matches among
    the teams from the given list of team IDs.
    Each key is a tuple (team1, team2), and the
    corresponding value is the string '   '. Each time a
    match is played between two teams, the corresponding
    entry should be updated with the result. Think of it
    as a large table (here with teams A, B, C):
       A  B  C
    A  x  x  x
    B  x  x  x
    C  x  x  x  (each x is a 3-space string)"""
    # use list comprehension
    matchboard_keys = [(a, b) for a in ids for b in ids]
    return dict.fromkeys( matchboard_keys, '   ' )
```

Figure 6.17: Program `premierleague_fncs.py`, part 1.

These short name IDs will be used as keys in several dictionaries, so they must be unique. Before I append my current short name `s` to the list, I need to make sure it's not already taken by another team. And if it is, I need to suggest another until I find one which isn't. That's what happens in lines 12–14: I have to stay here until my `s` is not in the list `shortnames`, hence the `while` in line 12. If the current `s` is in the list, I ask the user for an ID and assign it to `s`. I loop until `s` is not in the list. In line 15, the unique `s` is appended to the list, and in line 16, the list is returned.

Figure 6.17 also shows a function called `longestName` which takes a number, `length`, and a team tuple (name/strength again). It simply returns the maximum of the given length and the length of the team name `team[0]`. You'll see in a bit what it can be used for.

The next part of `premierleague_fncs.py` is shown in Figure 6.18: The function `playMatch` simulates playing a match between two teams with the given strengths `a` and `b`. One can think of zillions (well, many) ways to do that; to avoid messing with the upcoming, already much-hyped pretty-printing, I don't want any team to score more than 9 goals per game, so when I randomly choose the total number of goals scored in the match to be played (and call it `goals`) in line 42, I use `min` and `max` in suitable ways to force the number to be at most 9 and at least 0. The random number itself is picked through the `gauss` method of the `random` module; it returns a random number from a Gaussian distribution with the given mean and standard deviation (I use 3 and 2.5). In line 43, I initialize two goal counting variables `goals_a` and `goals_b`, and in line 44 I calculate the sum of the given strengths, `totalstrength`.

My idea is to loop `goals` rounds and in each round make a random but appropriately weighted decision as to who gets a goal. With probability $\frac{a}{a+b}$, the team with strength a scores a goal, and analogously, the other team scores with probability $\frac{b}{a+b}$. This is implemented by picking a random integer between 0 and `totalstrength` and comparing it to `a`; if the number is below `a`, team A scores (lines 46–47). Otherwise, team B scores. After the loop, I return a tuple containing the two goal counts.

The other function of Figure 6.18, `sorttuples`, is used to rank two teams according to their accumulated match statistics. It is defined to take two dictinary items `a` and `b`, each being a key/value tuple, as arguments (why this is so becomes clear in a little while). The key of each item is a team ID, the value is this team's 7-entry match statistics list as initialized in line 26 of Figure 6.16. The function is to be used by the function `sorted`, so it should return -1, 0 or 1 depending of the relative order of the given items. For readability, I introduce two variables `A` and `B` to point to the to match statistics lists.

If one team has more points than the other, I immediately return -1 or 1 (lines 61–64). If they have the same number of points, I look at the goal difference which can be calculated from the statistics tuple entries with indices 4 (goals scored) and 5 (goals received). If one of the teams has a better goal difference than the other, I can return -1 or 1, as appropriate (lines 65–68). As a last resort, if the goal differences are also equal, in lines 69–72 I look at which team has scored the most goals. If the two teams agree on this number as well, I return 0, indicating that their ranking can be arbitrary.

```
37 def playMatch(a, b):
38     """Play a match between two teams with the given
39     strengths. Return the tuple (ga, gb) of the number
40     of goals scored by a and b, respectively."""
41     # between 0 and 9 goals per match:
42     goals = int(min(9, max(0, gauss(3, 2.5))))
43     goals_a = goals_b = 0
44     totalstrength = a + b
45     for i in xrange(goals):
46         if randrange(totalstrength) < a:
47             goals_a += 1
48         else:
49             goals_b += 1
50     return goals_a, goals_b


51 def sorttuples(a, b):
52     """a and b are key/value dictionary items, i.e.
53     (id, T), where id is a team ID and T is its match
54     statistics tuple.
55
56     Return 1, 0 or -1 depending on whether a should come
57     before b when sorted accoring to the match
58     statistics of the two teams."""
59     A = a[1]
60     B = b[1]
61     if A[6] > B[6]:
62         return -1
63     elif A[6] < B[6]:
64         return 1
65     elif A[4]-A[5] > B[4]-B[5]:
66         return -1
67     elif A[4]-A[5] < B[4]-B[5]:
68         return 1
69     elif A[4] > B[4]:
70         return -1
71     elif A[4] < B[4]:
72         return 1
73     else:
74         return 0
```

Figure 6.18: Program `premierleague_fncs.py`, part 2.

Moving back to the main program, Figure 6.19 shows the tournament loop. I count the number of matches played in the variable `matches`. The loop itself is implemented as two nested `for` loops, each iterating through all IDs using the counter variables `teamA` and `teamB`, respectively (lines 31 and 32). This way, I'll get all possible pairs twice (both A, B and B, A). Of course, a team shouldn't play itself, so each time `teamA` is the same as `teamB`, I skip the loop body and move on to the next round (lines 33–34).

To call `playMatch`, I first need the strengths of the two teams. I get them through the `id2team` dictionary, using the IDs as keys: `id2team[teamA]` points to the team tuple of team A, the second item of which (index 1) is its strength. I retreive the strengths in lines 35 and 36 and then call `playMatch` in line 37, unpacking the tuple of goals scored in the match in variables `goalsA` and `goalsB`. Next, I create a result string (e.g. `'3-1'`) in line 38 and assign this string to the relevant entry in the match board dictionary (line 39). Recall that the keys in this dictionary are tuples of two team IDs.

```
27 # Play full tournament:
28 # The teams play each other twice.
29 # 3 points for a win, 1 for a draw, 0 for a loss

30 matches = 0
31 for teamA in ids:
32     for teamB in ids:
33         if teamA == teamB:
34             continue
35         strengthA = id2team[teamA][1]
36         strengthB = id2team[teamB][1]
37         goalsA, goalsB = playMatch(strengthA, strengthB)

38         resultstring = "%d-%d"%(goalsA, goalsB)
39         matchboard[(teamA, teamB)] = resultstring

40         if goalsA > goalsB:          # team A won
41             updateWinner(teamA, stats)
42             updateLoser(teamB, stats)

43         elif goalsA < goalsB:        # team B won
44             updateWinner(teamB, stats)
45             updateLoser(teamA, stats)

46         else:                        # draw
47             updateDraw(teamA, teamB, stats)

48         updateMatch(teamA, goalsA, teamB, goalsB, stats)
49         matches += 1
```

Figure 6.19: Program `premierleague.py`, part 2.

The rest of the loop of Figure 6.19 updates the statistics of teams A and B. If

team A won the match (`goalsA > goalsB`), I call the functions `updateWinner` and `updateLoser` on `teamA` and `teamB`, respectively, passing also `stats`, the pointer to the match statistics dictionary. If B won the match, I do the opposite. If the match was a draw, I call `updateDraw`. Finally, I call `updateMatch` and increment the `matches` counter. All these functions are listed in Figure 6.20.

```
75  def updateWinner(id, stats):
76      """The team given by the id has won a match;
77      update stats accordingly."""
78      stats[id][1] += 1  # matches won
79      stats[id][6] += 3  # points
80  def updateLoser(id, stats):
81      """The team given by the id has lost a match;
82      update stats accordingly."""
83      stats[id][3] += 1  # matches won
84  def updateDraw(idA, idB, stats):
85      """The two given teams have drawn a match,
86      update stats accordingly."""
87      stats[idA][2] += 1 # matches drawn
88      stats[idA][6] += 1 # points
89      stats[idB][2] += 1 # matches drawn
90      stats[idB][6] += 1 # points
91  def updateMatch(idA, goalsA, idB, goalsB, stats):
92      """Update stats for teams A and B according
93      to the match just played in which A scored
94      goalsA and B scored goalsB goals."""
95      stats[idA][0] += 1      # matches
96      stats[idA][4] += goalsA # goals scored by A
97      stats[idA][5] += goalsB # goals received by B
98      stats[idB][0] += 1      # matches
99      stats[idB][4] += goalsB # goals scored by B
100     stats[idB][5] += goalsA # goals scored by A
```

Figure 6.20: Program `premierleague_fncs.py`, part 3.

Recall that for a given team ID `id`, `stats[id]` points to a list of 7 values. These values are the number of matches played (index 0), matches won (in-

dex 1), matches drawn (index 2), matches lost (index 3), goals scored (index 4), goals received (index 5), and points (index 6). Thus, updating a match winner `id` means incrementing its "won matches" value, `stats[id][1]`, and adding 3 points to its "points total", `stats[id][6]`, as shown in lines 78 and 79 of Figure 6.20. Similarly, to update a loser, I increment its "lost matches" value (index 3, see lines 80–83), and to update two teams after a draw, index 2 ("drawn matches") is incremented for both, as well as their points scores (index 6); see lines 84–90. Finally, the `updateMatch` function updates the "goals scored" and "goals received" totals for both teams, as well as their "games played" value (lines 91–100).

Figure 6.21, then, shows the remainder of the main program which delivers the output. To print things in a pretty fashion, one needs to get down and dirty with counting characters, calculating column widths etc. That part in itself is not pretty and in fact rather tedious, but as you know, underneath a shining surface you often find an ugly truth!

I'm going to need a line of dashes (`----` etc.) of suitable length, so I calculate this length once and for all in line 51 of Figure 6.21. Then I move on to print the match board. I want it printed such that the team IDs appear alphabetically. To that end, I use the list method `sorted` to create a sorted copy of the `ids` list and assign it to the variable `sids` in line 54. I then pass this list to the string method `join`, using a space as the delimiter character. This creates a string with the sorted team IDs separated by one space each. I then print this string, prefixed by another three spaces (line 55) as the top row, the header, of the table.

Next, in alphabetical order, each team should be printed on its own row followed by the results of all the games it played. I again use the sorted list `sids` of IDs to iterate through the team IDs, both in the outer `for` loop in line 56 in which I first print the current team ID (line 57, note the comma yielding a space after the ID instead of a newline), and also in the internal loop (line 58). In the inner-most loop body, I simply print the result of the game played between the two current teams, `teamA` and `teamB`, looking up this result in the `matchboard` dictionary using the tuple `(teamA, teamB)` as key; again, the comma creates a space after the result (ensuring a total column length of four — this is why I didn't want any team to score more than 9 goals in one game..). After each internal loop, I end the row by printing just a newline (line 60).

Lastly, and most importantly, of course I want a table showing the final standings, including all the wonderful statistics. The columns of this table should be full team name and then the headers of each of the entries in the team statistics tuples. To really pretty-print it in a general manner, of course I need to *calculate* the width of the first column rather than just count the length of the longest team name, just in case I later change the list of teams playing the tournament. And strangely — what a coincidence — this high level of ambition leads me to use the `reduce` function introduced earlier.

What I need is the length of the longest name. The full team names are represented as the first items of the tuples in the `teams` list, remember? To go through all these items, keeping track of the maximum length seen so far,

```
50 # pretty-print match board:
51 dashes = (1+len(teams))*4
52 print "\nComplete match board (%d matches):"%matches
53 print "-"*dashes

54 sids = sorted(ids)
55 print "    ", " ".join(sids)
56 for teamA in sids:
57     print teamA,
58     for teamB in sids:
59         print matchboard[teamA, teamB],
60     print

61 # find length of longest team name:
62 namemax = reduce(longestName, teams, 0)

63 # create formatting string:
64 f = "%2d %2d %2d %2d %2d %2d %2d"

65 print "\nFinal standings:"
66 print "-"*dashes

67 print "Team%s  M  W  D  L Gs Gr  P"%(" "*(namemax-4))
68 print "-"*dashes

69 # sort teams by their statistics:
70 ranking = sorted(stats.iteritems(), sorttuples)

71 for id, (m, w, d, l, gs, gr, p) in ranking:
72     name = id2team[id][0]
73     print "%s%s"%(name, " "*(namemax-len(name))),
74     print f%(m, w, d, l, gs, gr, p)
```

Figure 6.21: Program `premierleague.py`, part 3.

I use `reduce`. I just need a tailor-made function which compares this current maximum with the length of the team name in the current team tuple, and that's exactly what the `longestName` function does (see lines 17–20 of Figure 6.17). So, the expression `reduce(longestName, teams, 0)` in line 62 of Figure 6.21 first calls `longestName` with the given starting value 0 and the first item from the `teams` list. Inside `longestName`, 0 is compared to the length of the string in index 0 of the item (which is a tuple), and the maximum of the two is returned. Next, `reduce` moves on to call `longestName` with this new, current maximum and the next item from the `teams` list. And so forth and so on. Eventually, the global maximum over all the list is returned by `reduce` and assigned to the variable `namemax`.

Recall that to print an integer of unknown size in a slot of fixed width, say 2, and print it right-oriented, you use the string conversion specifier `%2d`. In

line 64, I create a formatting string `f` in which I can later insert all the seven statistics values in right-oriented columns of width 3. For readability, I use `%2d` plus an explicit space rather than `%3d` for each column.

In lines 67–68 I print the header. For the first column, I need to add a number of spaces after the "Team" header, and I calculate this number as `namemax-4`. Then, in line 70, the decisive sorting of the teams takes place. I again use `sorted`, but this time I pass to it not only a list of items to be sorted but also a function telling it how to sort them. The list I want sorted is the list of key/value items from the `stats` dictionary, where the keys are team IDs and the values are the statistics tuples. I call the dictionary method `iteritems` to get the item list. You saw the sorting function, `sorttuples`, in Figure 6.18; recall that it ranks the dictionary items first by the team's points, then by goal difference, then by goals scored.

Calling `sorted` produces a new, sorted list of the dictionary items from `stats`. I can now iterate through this list in the `for` loop of line 71, unpacking each item into a key value (`id`), and a stats tuple (further unpacked into `(m, w, d, l, gs, gr, p)`). I need the `id` to look up the full name in the `id2team` dictionary in line 72 (that's why I didn't just sort the match stats tuples), and once I have that, I can print it followed by a suitable number of spaces, again calculated from the current name length and the overall maximum name length `namemax`. Next, in line 74 I use the formatting string `f` and insert all the individual values of the statistics tuple for the current team. Since the list is sorted, the tournament winner will appear at the top of the table.

Ah. I do think one might call this a tour de force of lists, tuples and dictionaries, eh compadre? Let's have a look at the output. If your enthusiasms for soccer and/or tables are weaker than mine, you might not find the output fully worth the effort ☺. To compensate, I'll print it in a happy green.

```
Complete match board (90 matches):
-------------------------------------------
    Ars AsV Bol Che Eve Liv MaC MaU Tot Wig
Ars     0-0 0-1 2-2 3-1 5-2 1-0 2-0 3-1 2-3
AsV 1-1     2-1 0-0 2-0 2-0 0-0 0-2 1-3 0-0
Bol 0-1 3-2     1-0 1-2 1-1 0-2 3-1 0-3 1-3
Che 0-3 4-0 0-0     3-3 2-3 0-0 5-0 0-0 5-1
Eve 0-0 0-0 1-0 1-2     2-2 2-2 2-2 1-2 1-1
Liv 3-1 0-0 1-3 0-0 2-2     4-3 2-0 4-0 2-2
MaC 0-0 0-2 1-3 0-2 1-2 1-2     0-0 1-0 0-1
MaU 2-1 0-0 0-0 3-2 3-3 0-0 1-0     4-0 1-0
Tot 0-1 0-0 0-1 1-5 2-2 0-0 0-0 0-2     3-1
Wig 0-1 0-2 1-3 1-6 0-0 3-2 2-1 0-1 2-1
```

As a random check, let's calculate the stats of Wigan, reading through the match board: 6 games won, 4 games drawn, 8 games lost, yielding 22 points. 21 goals scored, 32 goals received. We'll check those numbers in the final standings next, but first, note that, e.g. the calculated short version of the two-word name "Manchester United" is "MaU", as expected. Note also the

empty diagonal whose entries are strings of 3 spaces each — the default value given when constructing the `matchboard` dictionary in line 36 of Figure 6.17.

Conferring with the final standings, we see that the stats computed by hand for Wigan are correct:

```
Final standings:
-----------------------------------------
Team                M  W  D  L Gs Gr  P
-----------------------------------------
Arsenal            18  9  5  4 27 16 32
Manchester United  18  8  6  4 22 20 30
Chelsea            18  7  7  4 38 19 28
Bolton             18  8  3  7 22 21 27
Liverpool          18  6  8  4 30 27 26
Aston Villa        18  5  9  4 14 14 24
Wigan              18  6  4  8 21 32 22
Everton            18  3 11  4 25 28 20
Tottenham          18  4  5  9 16 28 17
Manchester City    18  2  6 10 12 22 12
```

Note that the width of the first column is indeed exactly the length of the longest team name plus 1. Note also that the ranking more or less mirrors the individual strengths of the teams.

While we're at it, having the whole machine up and running, let's play another tournament (a brownish one):

```
Complete match board (90 matches):
-----------------------------------------
    Ars AsV Bol Che Eve Liv MaC MaU Tot Wig
Ars     0-0 2-1 0-0 4-1 2-4 2-0 3-0 0-2 0-0
AsV 0-3     0-0 0-0 1-1 3-2 1-2 0-4 2-3 0-0
Bol 0-0 3-2     0-2 1-0 0-2 2-5 0-0 0-2 3-1
Che 1-0 0-3 0-0     2-1 1-2 3-1 1-4 0-0 5-1
Eve 0-2 1-0 0-2 3-0     7-2 0-2 0-0 3-1 0-0
Liv 1-2 0-1 2-1 3-2 6-0     0-0 2-0 3-2 3-3
MaC 1-2 0-1 0-0 1-1 3-4 1-2     0-0 1-4 0-0
MaU 0-0 3-0 1-1 1-0 0-0 4-2 2-3     0-0 1-0
Tot 1-2 0-0 2-1 2-5 4-4 0-5 0-0 1-1     0-1
Wig 0-0 2-1 1-1 0-0 0-0 0-0 3-2 1-3 2-3
```

In the standings calculated from this match board, we see the `sorttuples` function fully blossoming. This tournament was won by Liverpool but only at the narrowest margin possible: Arsenal and Liverpool both ended up with 33 points, and they both have a goal difference of $+12$. Thus, the impressive 41 goals scored by Liverpool is what gives them the trophy, as dictated by lines 69–72 of the `sorttuples` function in Figure 6.18, since Arsenal only scored 24. Thus, Liverpool avenge the bitter defeat suffered back in 1989.

```
Final standings:
-----------------------------------------
Team                 M  W  D  L Gs Gr  P
-----------------------------------------
Liverpool           18 10  3  5 41 29 33
Arsenal             18  9  6  3 24 12 33
Manchester United   18  7  8  3 24 14 29
Chelsea             18  6  6  6 23 22 24
Tottenham           18  6  6  6 27 30 24
Everton             18  5  6  7 25 30 21
Bolton              18  4  7  7 16 22 19
Wigan               18  3 10  5 15 22 19
Manchester City     18  4  6  8 22 27 18
Aston Villa         18  4  6  8 15 24 18
```

This program has painstakingly exhibited the strengths of sequences and dictionaries and their related functions. It also demonstrates what lists should not be used for: The whole bookkeeping necessary to keep track of the meaning of the seven values in the statistics tuples of each team is really intolerable. That the number of goals scored is located in index 4 is a fact very easily forgotten or confused, and this whole data structure is very obscure indeed. It would be much better if we could reference the "goals scored" value by some meaningful name rather than by some meaningless index. In the next chapter, you'll see how we can do that.

Lastly, note how I through the whole program exploit the fact that arguments passed to functions are references to data, not copies of data. For example, each time I pass the dictionary `stats` to one of the update functions of Figure 6.20, the updates are performed on the *original* `stats`, not some local copy. Thus, they "survive" when the function call terminates, and that is exactly the behavior I need.

The 1988/1989 season match between Liverpool and Arsenal had been postponed until the very end of the season, and in the meantime Liverpool had won the FA Cup, meaning they had the chance of a historic second Double. Before the match, Arsenal were on 73 points with 71 goals scored and 36 received (meaning a goal difference of +35); Liverpool were on 76 points with 65 goals scored and 26 received (a difference of +39). Thus, Arsenal needed to win by at least two goals to take the title: That would level the teams at 73 points and goal difference +37 but with Arsenal having scored more goals than Liverpool. Liverpool had not lost by two goals at home at Anfield for nearly four years.

After a goalless first half, Alan Smith scored soon after the restart, heading in a free kick from Nigel Winterburn. As the clock ticked down, though, it looked as if Arsenal were going to win the battle but lose the war. However, in injury time, in Arsenal's last attack, Michael Thomas surged from midfield, ran onto a Smith flick-on, evaded Steve Nicol and shot low past Bruce Grobbelaar to score Arsenal's second, and win the title, Arsenal's first in eighteen years. (Source: `http://en.wikipedia.org/wiki/Michael_Thomas`).

Please indulge me — here's a final one, sorry. It won't be in the exam.

```
Complete match board (90 matches):
------------------------------------------
    Ars AsV Bol Che Eve Liv MaC MaU Tot Wig
Ars     2-2 3-3 0-0 4-1 0-0 1-0 0-2 1-1 2-1
AsV 1-3     0-0 0-0 0-0 0-2 0-1 2-0 2-4 1-0
Bol 0-1 0-0     0-0 0-0 0-0 2-1 1-1 2-4 2-0
Che 1-3 6-3 3-0     0-0 3-1 0-0 0-1 4-1 0-0
Eve 0-0 1-0 1-1 1-5     2-2 3-0 1-1 2-0 4-1
Liv 1-2 3-2 0-1 0-0 4-1     0-0 2-1 0-0 3-1
MaC 0-6 0-0 0-0 1-1 0-0 0-0     0-0 3-6 1-0
MaU 0-4 2-1 0-1 1-1 5-0 1-0 2-1     2-1 4-0
Tot 0-0 0-0 0-1 0-0 2-1 1-0 1-0 0-0     1-1
Wig 1-4 1-2 1-1 0-3 2-3 2-4 1-4 1-3 1-2


Final standings:
------------------------------------------
Team               M  W  D  L Gs Gr  P
------------------------------------------
Arsenal           18 10  7  1 36 14 37
Manchester United 18  9  5  4 26 16 32
Chelsea           18  6 10  2 27 12 28
Tottenham         18  7  7  4 24 20 28
Liverpool         18  6  7  5 22 17 25
Bolton            18  5 10  3 15 15 25
Everton           18  5  8  5 21 27 23
Manchester City   18  3  8  7 12 23 17
Aston Villa       18  3  7  8 16 25 16
Wigan             18  0  3 15 14 44  3
```

Randomness truly is wonderful: We are blessed with an extraordinary outcome! Not only does Tottenham's remarkable $6-3$ win against Manchester City help them to a healthy 4th place in the standings despite their low strength of only 6; we may also wonder at the poor fate of Wigan winning no game in the entire season, losing all but 3!

That's all for the sports this evening. Thanks for watching.

Now you know about:

- Lists and tuples
- Unpacking
- Mutable vs. immutable
- The `del` command
- The `xrange` function
- List methods
- `enumerate`, `zip`, `map`, `reduce` and `filter`
- List comprehension
- Sets
- Dictionaries
- Dictionary methods
- The `deepcopy` function

Chapter 7

Classes and objects

Classes and objects are the fundamental units of *object oriented programming*. When dealing with real-world data, the natural data unit is usually not simply a number, a string or a list. E.g., to represent a bank account, a protein or a soccer tournament, you need to be able to pack together more, and more complex, information. A bank account is more than the balance; it also has a history of withdrawals and deposits. A protein is more than a string holding its amino acid sequence; e.g., it has a 3D structure. A soccer tournament is more than a list of team names; each team has a record of won and lost matches, there's a match board, etc. This chapter shows you how to model such phenomena.

7.1 Compound data

Imagine you want to build a database over DNA sequences from various animal species. Each data item should contain the sequence and the name of the species, and let's just say we want to keep the items in a list.

Figure 7.1 shows one way of managing the data. Three times it reads a species name and a DNA sequence from the user, puts them in a tuple and appends the tuple to a list. Thus we end up with a list of tuples. To look up an item, we first index the `data` list to get to the right item, and then index this item to get to its sequence (which has index 1 in the data item tuple), as shown in line 7.

```
1 data = []
2 for i in range(3):
3     species = raw_input("Species name: ")
4     sequence = raw_input("DNA sequence: ")
5     data.append( (species, sequence) )

6 # look up a data item:
7 print "The sequence of the second item is",data[1][1]
```

Figure 7.1: Program `data_compound_tuple.py`.

The expression `data[1][1]` is not very informative as to what exactly we're getting. One has to remember that the sequence is the second member of each data item tuple, and that the species name is the first. If we query the user for a species to look up, there's a nice way to remedy that, however, so let's modify the program slightly:

```
1  data = []
2  for i in range(3):
3      species = raw_input("Species name: ")
4      sequence = raw_input("DNA sequence: ")
5      data.append( (species, sequence) )
6  # look up a data item:
7  query = raw_input("\nLook up which species? ")
8  for species, sequence in data:
9      if species == query:
10         print "The sequence of %s is:"%query
11         print sequence
12         break
```

Figure 7.2: Program `data_compound_tuple2.py`.

Now we ask the user for a species to lookup, the `query` in line 7. Next, in line 8, we iterate through the `data` list *unpacking* each data item tuple as we go. Giving each tuple member a meaningful name (`species` and `sequence`) is better than having to refer to them by meaningless indices. Inside the loop we check if the current `species` matches the `query`; if it does, we print the associated sequence and `break` out of the loop — if you find what you're looking for in a list, don't waste time going through the rest of the list. Output from a run is shown in Figure 7.3.

```
1  Species name: dog
2  DNA sequence: CAGCTATCGGCT
3  Species name: cat
4  DNA sequence: CTAGCCGATATTC
5  Species name: porcupine
6  DNA sequence: TAGAGACCCCA
7
8  Look up which species? cat
9  The sequence of cat is:
10 CTAGCCGATATTC
```

Figure 7.3: Output from program `data_compound_tuple2.py`.

Now, let's expand our data items a bit. Let's add the Latin name to each data item so that it becomes a tuple with three members rather than just two.

That means we also have to change the look-up code; otherwise, the unpacking of values from each tuple goes wrong since you can't unpack three members of a triple into two variables:

```
1  data = []
2  for i in range(3):
3      species = raw_input("Species name: ")
4      latin = raw_input("Latin name: ")
5      sequence = raw_input("DNA sequence: ")
6      data.append( (species, latin, sequence) )

7  # look up a data item:
8  query = raw_input("\nLook up which species? ")
9  for species, lat, sequence in data:
10     if species == query:
11         print "The sequence of %s (%s) is:"%(query, lat)
12         print sequence
13         break
```

Figure 7.4: Program `data_compound_tuple3.py`.

What if we later decide to add more information to each data item? For example, we might later repeatedly need the length of a sequence (i.e. the number of nucleotides it contains), and so rather than calculating every time we need it, we might as well calculate it once and for all and put it in the tuple together with the name, the latin name, and the sequence itself.

The unpacking solution is not sustainable; it depends directly on the order and number of members of the data item tuples. With suddenly a length in each tuple as well, we'd have to add an unpacking variable to line 9 of Figure 7.4. It's tedious and complex and not what we want. We need some way of putting all the associated information into one unit, one compound of data, which we can freely expand later.

You can think of a *class* as a blueprint, a how-to for building something. The things you build from a class are called *objects*. A class may describe objects that contain various components, e.g. data and functions manipulating this data. It may also just contain generally useful functions. All objects created from the same class will have the same structure and the same components. Class/object data are called *attributes*, class functions are called *methods* because they describe precisely which methods are available for dealing with the data.

We have already met at least three classes, namely `str`, `list` and `dict`. An object created from each of these contain data (a string, a list and a dictionary, respectively), and the functions they provide manipulate this data. That's why I referred to them as *string methods*, *list methods* and *dictionary methods* rather than functions: They are actually methods defined inside a class.

In our running example, we might create a `Seq` class to hold all the information needed for each data item. You don't have to put anything in the class

initially; you're allowed to add stuff to your objects as you go along. As an initial example, consider this definition:

```
1  >>> class Seq:
2  ...   pass
3  ...
4  >>> a = Seq()
5  >>> a.name = "cat"
6  >>> a.latinname = "felix domesticus"
7  >>> a.sequence = "AGCGATATGCTAG"
8  >>> print a.name, a.latinname, a.sequence
9  cat felix domesticus AGCGATATGCTAG
10 >>>
11 >>> b = Seq()
12 >>> print b.name
13 Traceback (most recent call last):
14   File "<stdin>", line 1, in ?
15 AttributeError: Seq instance has no attribute 'name'
```

In line 1, you see the notation used to define a class. It is simply the keyword `class` followed by a legal name followed by a colon. Then, an indented code block should follow. If you put nothing into your class a priori, you can simply put a `pass`.

To create (*instantiate*) an object (*instance*) from the class, simply write the class name followed by parentheses, as shown in line 4. This creates an object and a reference to it, and usually you then need to assign this reference to some variable to have a handle to the object.

Creating an object creates a local namespace belonging to that object. To get access to the inside of an object, i.e. to its namespace, you use the dot operator, just as you do when calling a string method on some string. Thus, to add variables (object attributes) `name`, `latinname` and `sequence` to the object `a` just created, you simply set them by dotting your way inside `a`, as shown in lines 5–7. The variables are created inside the namespace of the particular instance `a` of the `Sequence` class; thus, when we create a second instance `b` in line 11 and attempt to access its `name`, we get an error message: `b` doesn't have this attribute.

As explained above, one of the good things about a class is that its objects all have the same structure; i.e., the structure laid out in the class definition. Obviously, with an empty class definition this benefit is lost. We wish to enforce that *all* `Seq` objects have a name, a Latin name and a sequence, otherwise they are not real `Seq` objects. This can be done by restricting the way objects can be instantiated.

When you instantiate an object, what happens is that Python looks inside the class definition for a so-called *constructor* method. If the class doesn't have one, Python equips it with a default, empty one requiring no arguments. The

constructor method defines how objects of the class may be instantiated and performs initialization of the object if any is needed. The constructor has to be called `__init__` (the double underscores in front and at the end are hideous, I know, but they're there to make it clear that this is a special method), and it has to return `None` (or have no `return` statement which is equivalent). Thus, if we want all `Seq` objects to have the attributes `name`, `latinname` and `sequence`, we can use a constructor to enforce it. Figure 7.5 shows the new class definition.

```
1 class Seq:
2     """A class for holding information about a DNA
3     sequence"""
4     def __init__(self, n, l, s):
5         self.name = n
6         self.latin = l
7         self.sequence = s.upper()
```

Figure 7.5: Program `data_compound_class.py`.

Let me start with the easy stuff: The string in lines 2 and 3 is a documentation string: It's good programming practice to add documentation strings to your function, method and class definitions which describe what they do. A documentation string must appear right after the definition and should be in triple quotes.

In line 4, the constructor is defined. It takes four parameters called `self`, `n`, `l` and `s`; forget about the `self` for a moment. As you know, parameters of a function are local variables to that function, so once the `__init__` method terminates, they disappear. In the previous example above, we used the dot operator to point to a specific object instance "from the outside" to put attributes inside it, but here, in the constructor, we're "inside it", so how can we save them in the object being created? How can an object point to itself? It can using the `self` parameter which must be the first parameter in *all* methods of a class. When Python creates an object, it allocates a lump of memory, creates a reference to it — and passes this reference as the first argument to the constructor. That's how an object can point to itself. In lines 5–7, therefore, we can access the namespace of the object just created through the `self` parameter and create three attributes `name`, `latin` and `sequence`, assigning them the values of `n`, `l` and `s.upper()`, respectively (we wish to keep our sequences in uppercase letters). When the constructor terminates, `n`, `l` and `s` go out of scope, but the object lives on, and their values are saved inside it under the attribute names `name`, `latin` and `sequence`.

It's often better to use class definitions from other programs by importing them; that way they can easily be reused without the need to remove any main program later. So, we might test it like this:

```
1  >>> from data_compound_class import Seq
2  >>> a = Seq("cat", "felix domesticus", "CTAGCTACG")
3  >>> print a.name, a.latin, a.sequence
4  cat felix domesticus CTAGCTACG
5  >>>
6  >>> b = Seq("cat", "felix domesticus", "CTAGCTACG")
7  >>> a == b
8  False
```

In line 2, we instantiate an object of the `Seq` class. This time, we have to give it some arguments as dictated by the constructor — an instantiation like `a = Seq()` with no arguments would result in an error. Note that we *don't* supply an argument for the `self` parameter as the object reference is provided by Python automatically. It's probably confusing that any class method must take at least one argument because of the `self`; in other words, any method of any class takes n arguments where $n \geq 1$, but when you *call* it you supply only $n - 1$ arguments. Personally, I find that extremely user-unfriendly, but alas, that's how it is. Let me repeat this in a bulleted list:

- *Any* method of *any* class must take at least one argument.
- By convention, the first parameter of all class methods is called `self` to indiciate that it points to the object itself.
- You *don't* supply the first argument from the method definition when calling the method.

Lines 3 and 4 show that the arguments we supplied when instatiating the object are actually saved inside it and can be retrieved using the object name and the dot operator. Line 6 shows that two objects are not considered equal even if they contain identical data: Object references are only equal if they point to physically the same object.

As I mentioned, besides determining how objects should be instantiated, the constructor is also the obvious place to perform any initializations. If we know that we'll need the length of the given sequence numerous times in the future, why not calculate it once and for all and store it in the object? It's quicker and more elegant to look up something than to recalculate it. We might as well let the constructor handle it for us.

Figure 7.6 shows an extended version of the `Seq` class. The only change is line 8 where the length of the given sequence is calculated and stored in an object attribute called `length`. Note that it is calculated based on the arguments already given; we still instantiate `Seq` objects exactly the same way, only they now hold more information. We also access the name, latin name and sequence data in exactly the same way as before, even though the class definition has changed. Any programs already using the class will still work (that's called *backwards compatibility*). Isn't that beautiful?

```
1  class Seq:
2      """A class for holding information about a DNA
3      sequence"""
4      def __init__(self, n, l, s):
5          self.name = n
6          self.latin = l
7          self.sequence = s.upper()
8          self.length = len(s)
```

Figure 7.6: Program `data_compound_class2.py`.

To exhibit the weaknesses of the programs in Figures 7.2 and 7.4, Figure 7.7 shows a program which utilizes the `Seq` class. Collecting the data from the user takes place in the same way, but instead of creating a tuple holding each data item and appending it to the `data` list, an object is created and appended (line 7). In this example, the reference to the object is not assigned to a variable but rather stored in a list, but since the list is accessible through the variable `data`, so is the object (by indexing into or iterating through the list).

```
1  from data_compound_class2 import Seq
2  data = []
3  for i in range(3):
4      species = raw_input("Species name: ")
5      latin = raw_input("Latin name: ")
6      sequence = raw_input("DNA sequence: ")
7      data.append( Seq(species, latin, sequence) )
8  # look up a data item:
9  query = raw_input("\nLook up which species? ")
10 for item in data:
11     if item.name == query:
12         print "%s (%s):"%(item.name, item.latin)
13         print "%s (%d nt)"%(item.sequence, item.length)
14         break
```

Figure 7.7: Program `data_compound_class_main.py`.

We iterate through the items in the `data` list in line 10. In each round of the loop, `item` now points to an *object*. To compare its name attribute with the query, we simply use the dot operator to get to `name` (line 11). If we find a match, we print its information — including its length which we now automatically have since the constructor calculated it for us.

Figure 7.8 shows essentially how the memory looks just before line 9 of the program is executed. The global namespace contains, among others, the two

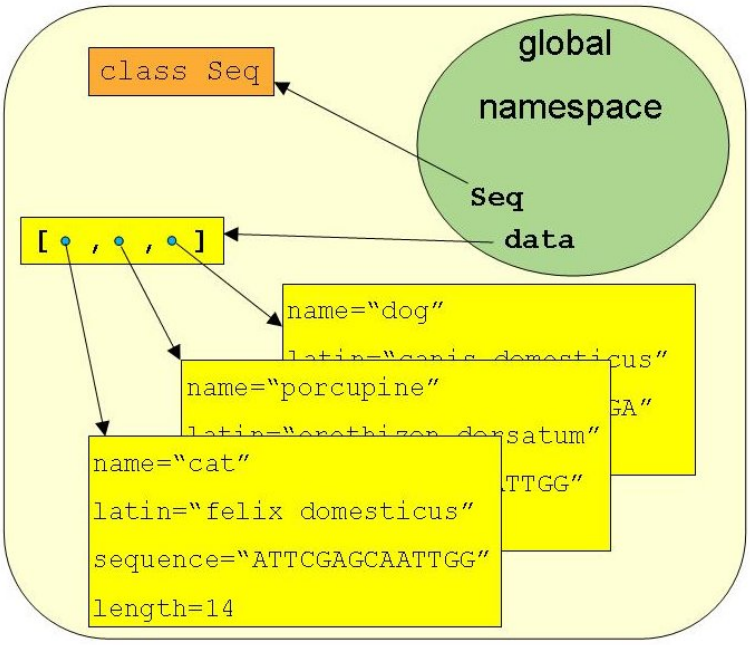


Figure 7.8: A look at the memory during the execution of the program in Figure 7.7.

```
Species name: cat
Latin name: felix domesticus
DNA sequence: ATTCGAGCAATTGG
Species name: porcupine
Latin name: erethizon dorsatum
DNA sequence: TACGCATATA
Species name: dog
Latin name: canis domesticus
DNA sequence: ATGCTAGCATGA

Look up which species? cat
cat (felix domesticus):
ATTCGAGCAATTGG (14 nt)
```

Figure 7.9: Output from program `data_compound_class_main.py`.

identifiers `Seq`, which points to the class definition, and `data`, which points to a list. This list in turn has three members which are merely unnamed references to objects. The objects are "structurally" identical in that they all have four local attributes called `name`, `latin`, `sequence` and `length`, but *they all have their own set of values* for these attributes. A trial run is shown in Figure 7.9.

I hope this section has convinced you that classes are great for *encapsulating data*. Rather than stuffing everything into lists of lists of lists which are

impossible to maintain and navigate in, you can bundle your data in a nice compound with proper, meaningful names by defining some class for it. In the next section, we take the idea a little further.

Erethizon dorsatum means "to irritate with your back" in Latin. Source: http://www.nativetech.org/quill/porcupin.html

7.2 Abstract data types

You know integers. Integers are a certain kind of numbers. There are certain things you can do with integers: You can add them to each other, you can subtract them from each other, etc. The Python integer *type* is an example of a simple data type, but one can easily imagine more complex types.

You might think of our `Seq` class of the previous section as an *abstract data type*, sometimes called a *data structure*; i.e. a special data type whose values are abstract representations of something. In this case, each member of the `Seq` type is a DNA sequence. As with the integers, any abstract data type should define what operations one can perform on them. That's done in a class by adding methods to it which manipulate its data.

First of all, the way we have accessed the attributes up until now — dotting our way to them directly through the dot operator — is not perfectly kosher. A proper data type has *access methods* which are simply methods that return the data which the user should have access to. If the user only accesses the content of an object through its access methods, the class programmer controls what data should be visible to the user and what shouldn't. Or at least, the class programmer can *signal* to the user what data should be accessed and what shouldn't — all attributes are visible from outside the objects through the dot operator, so the user may cheat and dot his way through to access some attribute which the programmer hasn't provided an access method for, but convention dictates that users should respect the class author's intentions and not access any attributes besides through the access methods.[1]

Here are some access methods for the `Seq` class:

```
    def getName(self):
        return self.name

    def getLatinname(self):
        return self.latin

    def getSequence(self):
        return self.sequence

    def getLength(self):
        return self.length
```

[1]This is another slightly silly feature of Python; there's no way of really hiding class content from users, unlike, e.g., the *private* attributes of Java. You can obstruct access, but since you can't actually prevent it, I won't go further into this subject.

An access method is simply a method with a self-explanatory name which returns the value of some object attribute. Here, again, we see the `self` parameter. The access methods should be *called* with no arguments, but even so each method definition takes *one* parameter.

The methods of a class only exist in one copy; they belong to the local namespace of the *class*, and each object instantiated from the class is *not* given its own copy of all the methods. So when you call the `getName` method of the `Seq` class via some particular object, how does Python, from within the method in the class definition, get to the right object in which to look up the `name`? By necessity, the method needs some pointer to the object, and it gets this pointer from the method call:

```
item.getName()
```

The object referenced by the identifier `item` is the object whose name we want. *This object reference is automatically transferred as the first argument to the method.* That's how the class method knows which object it should deal with: The first parameter, the `self`, holds a reference to it, and inside the method this `self` is used to access the particular object's local namespace. And therefore *all* class methods must take at least one argument; otherwise they couldn't manipulate any object specific data.

Usually in abstract data types, you also supply a suite of methods for *setting* or modifying the attributes. In our running example, we might include a `setSequence` method — for the sake of illustration — just in case an updated DNA sequence for some animal becomes available:

```
    def setSequence(self, s):
        self.sequence = s
        self.length = len(s)
```

Of course, this method needs an explicit argument, a new sequence to assign to the `Seq` object. Thus, the *definition* takes a second parameter `s` besides the mandatory `self`, and when you *call* it, you call it with *one* argument. Inside the method, the given `s` is assigned to the referenced object's `sequence` attribute — and the `length` attribute is updated accordingly to keep the state of the object consistent.

One of the reasons why such access methods should be provided is that they promote robustness. If any reference to object data, including from inside the object, goes through the access methods, then the programmer can easily change how data is managed in the class — he[2] only has to update the access methods and needn't worry about programs using the class. If some program somewhere imports the class and refers directly to, e.g. the `latin` attribute, that program will fail if the `latin` attribute is renamed or removed. If the program instead only refers to it through its access method, the class programmer can make sure a correct, or at least meaningful, response is given.

[2]Throughout this text I will use "he" as a shorthand for "he or she".

Any interesting abstract data type can have several values or instances, and as soon as several values are in play, it becomes natural to want to be able to compare them. For numbers it is obvious what it means to compare; for an abstract data type, such a measure for comparison is often neither given nor evident. In other words, it is up to the programmer to devise a way of comparing two data type values.

Given two `Seq` instances, what's to compare? We might compare their names (but why would that be interesting?) or their lengths (even less interesting). What would make sense, though, would be to measure the *evolutionary distance* between the two DNA sequences. Here, we'll use a very simple model, namely a variant of the *Hamming distance* (see `http://en.wikipedia.org/wiki/Hamming_distance`):

1. Given two DNA sequences of the same length, the Hamming distance between them is the number of positions where the corresponding nucleotides differ.

2. If the sequences are of different lengths, their Hamming distance is the minimum Hamming distance one can achieve by comparing the shorter sequence with similar-length slices of the longer sequence.

For example, the Hamming distance between `ACGGAAT` and `ATGGGCT` is 3:

```
ACGGAAT
ATGGGCT
```

because the sequences differ in three positions (shown in red), while the Hamming distance between `TTACGATG` and `ACGAG` is 1:

```
TTACGATG
  ACGAG
```

because the minimum number of differing positions you can get by any alignment of the shorter sequence to the longer one (ignoring extra nucleotides at the ends, shown in grey) is 1.

All living things are descendants from the same ancestor (*way* back). All through history, living organisms have used the same DNA language to encode their genetic material. By minor modifications such as mutations and rearrangements, the DNA of an organism may be slightly different from that of its offspring. As evolution progresses, these differences gradually grow large enough to form new species. Thus, one way of tracing the evolutionary history of species is to compare their DNA; the DNA of close relatives is very similar, while the DNA of more distant relatives is less similar.

Figure 7.10 shows the `Seq` methods which implement the Hamming distance measure. The `distanceTo` method is the "real" distance method while `d_equal_Lengths` is a helper method. The way to compare `Seq` objects `a` and `b` is to let one of them compare itself to the other, i.e. to call the `distanceTo` method of one object with the other object as argument: `a.distanceTo(b)`. As the measure is symmetric, it doesn't matter which object plays which role.

Both methods have documentation strings; the access methods don't since their names are self-explanatory. To keep the code readable, in lines 4–7 the sequences and lengths of the `Seq` objects are retrieved once and for all using their

access methods. Note that the call `s.getSequence()` would fail if `s` were not a `Seq` object; as Python supports what is called *dynamic typing*, it doesn't check for you that the actual argument transferred to a method or function is of the right type. That is completely your own responsibility.

```
1     def distanceTo(self, s):
2         """Return the Hamming distance between this
3         Seq object and the given object s"""
4         s1 = self.getSequence()
5         l1 = self.getLength()
6         s2 = s.getSequence()
7         l2 = s.getLength()
8         if l1 == l2:
9             return self.d_equalLengths(s1, s2)
10        if l2 > l1:
11            # make sure s1 is the longest sequence
12            l1, l2 = l2, l1
13            s1, s2 = s2, s1
14        d = l1  # initialize to maximal value
15        for i in range(l1-l2+1):
16            s1slice = s1[i:i+l2]
17            d = min(d, self.d_equalLengths(s2, s1slice))
18        return d
19    def d_equalLengths(self, s1, s2):
20        """Return the Hamming distance between the
21        two strings of equal length, s1 and s2"""
22        m = 0
23        for i in range(len(s1)):
24            if s1[i] != s2[i]:
25                m+=1
26        return m
```

Figure 7.10: The Hamming distance methods from the program `abstract_datatype.py`.

If the two involved sequences have the same length, the helper method `d_equal_Lenghts` is called (line 9) on them (the *string sequences*, not the objects). `d_equal_Lenghts` simply counts the number of positions where its two parameter strings differ, assuming that they have the same length. After initializing a counter, `m` (in line 22), the `for` loop of line 23 iterates through the indices of the strings, and each time the nucleotides in index `i` of the two strings are not the same, `m` is incremented (lines 24–25). Eventually, `d_equal_Lenghts` returns the result in line 26, and in turn, this result is passed on and returned

by the `distanceTo` method in line 9.

If the two strings do not have the same length, we need to try all possible alignments of the shorter one to the longer one. To do that, we need to know which is which. Deciding that `s1` should be the longer one, we swap the lengths and sequence strings in lines 12–13 if that is not a priori the case.

Going through the alignments, we use the variable `d` to store the minimal Hamming distance seen so far. In line 14, it is initialized to some large dummy value. The loop in lines 15–16 tries all possible alignments of the shorter string `s2` to a slice of the longer string `s1` of equal length, calculating the Hamming distance of each alignment using the helper method, and updates `d` to the minimum of this distance and `d`'s current value. Resetting `d` this way ensures that `d` stays the current minimum.

In more detail: Given the two strings `s1` and `s2` of lengths `l1`> `l2`, how many alignments do we need to perform? The answer is (`l1-l2+1`) as illustrated in this example where we get 3 alignments with `l1`=5 and `l2`=3:

```
ACAGT     ACAGT     ACAGT
GGG        GGG        GGG
```

Hence, the `for` loop in line 15 runs `l1-l2+1` times. In each round, `s2` should be compared to a new slice of `s1`. The loop counter variable `i` is the start index of this slice. In the first round, we should compare `s2` to `s1[0:l2]`, in the next we should compare `s2` to `s1[1:l2+1]`, etc. We retrieve the right slice, `s1slice`, in line 16.

Now we have two strings of equal length: `s2` and `s1slice`. We compare them with the helper method in line 17. If the result is less than the current minimum `d` (this test is performed by the `min` function, also in line 17), `d` is updated accordingly. When the loop terminates, `d` is returned.

```
1  if __name__=="__main__":
2      data = []

3      # create three objects:
4      for i in range(3):
5          species = raw_input("Species name: ")
6          sequence = raw_input("DNA sequence: ")
7          data.append( Seq(species, None, sequence) )

8      # calculate their Hammond distances:
9      for s1 in data:
10         for s2 in data:
11             n1 = s1.getName()
12             n2 = s2.getName()
13             h = s1.distanceTo(s2)
14             print "H(%s, %s) = %d"%(n1, n2, h)
```

Figure 7.11: Testing the Hamming distance method.

Figure 7.11 shows a test program — I got tired of typing in Latin names, so I changed it a bit and set each Latin name to `None` — and Figure 7.12 shows a trial run. Note that each `Seq` object is also compared to itself and that the result in such a case correctly is 0. As a check, with a little effort we find that the Hamming distance between porcupine and dog is indeed 3, as predicted by the program, since in one possible alignment of the dog and porcupine sequences, we get 2 out of 5 matches (and thus a difference of 3), while in the others we have no matches at all (and thus differences of 5):

```
CCCAC        CCCAC        CCCAC
                             ||
AATGGAC     AATGGAC     AATGGAC
```

```
Species name: cat
DNA sequence: attcgac
Species name: dog
DNA sequence: aatggac
Species name: porcupine
DNA sequence: cccac
H(cat, cat) = 0
H(cat, dog) = 2
H(cat, porcupine) = 2
H(dog, cat) = 2
H(dog, dog) = 0
H(dog, porcupine) = 3
H(porcupine, cat) = 2
H(porcupine, dog) = 3
H(porcupine, porcupine) = 0
```

Figure 7.12: Output from program `abstract_datatype.py`.

You saw in Section 7.1 that a class can encapsulate raw data. In this section, you've seen the proper way of doing it using access methods, and you've seen that a class can also encapsulate *function*. The programmer defines what can be done with the encapsulated data through the methods he provides; these methods constitute the *interface* to the class.

7.3 Recursive data types

If we have a whole set of DNA sequences, represented as `Seq` objects, we might begin wondering about their evolutionary relationships. We know how to calculate (a primitive approximation to) the evolutionary distance between two sequences; can we use this to build a phylogeny over the sequences? How can we even represent a binary tree structure in Python?

The thing about binary trees is that they come in different sizes. A binary tree may have any depth, so it seems two trees don't have to have the same structure at all. How, then, can we model a tree as a class in Python?

Well, we can note that all *nodes* in a tree *do* have the same structure: Each node has a parent node, it holds some data, and it has two child nodes (except for the root which doesn't have a parent node, and the leaves which don't have child nodes). Viewed this way, a tree is either empty or a node with two child trees. What I'm getting at is that we need a *recursive data type*, a class which allows its instances to point to other instances of the same class. Figure 7.13 shows such a class called `Node`. Most of it is stuff you've seen before, but there are some nifty new features.

A *phylogeny* is a represention of the evolutionary relationships between a group of organisms. It is a *binary tree*, i.e. an upside-down tree consisting of interconnected *nodes* corresponding to the organisms. The *root* node is at the top and the *leaves* are at the bottom, and all nodes in the tree have two descendants (also called *children*) except the leaves which have none. The *depth* of a tree is the length of the maximal path from the root to any leaf node.

First of all, line 4 creates a *class attribute*. Unlike object attributes, a class attribute belongs in the namespace of the *class* and thus only exists in one copy. You can access a class attribute either through the class name, as we shall see, or through objects of the class like with object attributes and object methods, but in any case you'll be referencing the one, *same* attribute. Information pertaining to the entire class can be stored in a class attribute, like e.g. the total number of objects instantiated from the class. Here, `id` is a number which is used to give each new node a unique ID (initialized to 1).

The constructor of the `Node` class (lines 5–11) takes four regular parameters besides the `self`. The parameter `d` should be a reference to some data item, and it is stored in the object attribute `data`. The `pnode` should be a reference to the parent node of the node currently being created, and it is stored in the attribute `parent`. The next two parameters, `lnode` and `rnode`, are references to the left and right child nodes of this node, respectively. Just in case they don't yet exist when a new node is created, the parameters are defaulted to `None` so that they point to nothing. The first two parameters don't have default values: The user must *explicitly* provide some data item and a parent node when creating a new `Node` instance.

In line 10, I assign a unique ID to this new node using the class attribute `id`, accessing it through the class with `Node.id`. A class attribute is created and initialized only once — when Python reads the class definition — and since it is defined outside the constructor, it is *not* reinitialized in every subsequent object instiantiation. Thus, the very first node will get the ID `"N1"` because `Node.id` was initialized to 1. In line 11, I update the class attribute so that the next `Node` instance will get a new ID. Accessing a class attribute through the class name clearly signals that it belongs to the class, not an object, although as mentioned you can also access class attributes through individual objects.

The methods in lines 12–27 are access methods for retrieving or (re-)setting attribute values. The three methods `setLeft`, `setRight` and `setParent` tacitly expect a `Node` instance as parameter, while the user may assign any data item to the node with the `setData` method — I intend to keep the `Node`

```
1  class Node:
2      """Represents a node in a binary tree with references
3      to its parent, two child nodes and a data item."""
4      id = 1       # used to assign unique ID to each node
5      def __init__(self, d, pnode, lnode=None, rnode=None):
6          self.data = d
7          self.parent = pnode
8          self.left = lnode
9          self.right = rnode
10         self.id = "N%d"%Node.id  # give ID to new node
11         Node.id += 1
12     def getLeft(self):
13         return self.left
14     def getRight(self):
15         return self.right
16     def getParent(self):
17         return self.parent
18     def getData(self):
19         return self.data
20     def setData(self, d):
21         self.data = d
22     def setLeft(self, node):
23         self.left = node
24     def setRight(self, node):
25         self.right = node
26     def setParent(self, node):
27         self.parent = node
28     def __str__(self):
29         return self.id
```

Figure 7.13: Program `binary_tree.py`.

class general and allow it to contain any kind of data, not just `Seq` objects.

Finally, in lines 28–29 a weird method sees the light of day: `__str__`. Like the two other underscore-ridden identifiers you know, `__init__` and `__name__`, this guy is a general thingy which one may or may not include in any class definition. IF it's there, it is called automatically each time the user tries to

create a string representation of object, e.g. if he tries to `print` it directly. If you `print` an instance of a class which doesn't have a `__str__` method, you'll get some technical stuff including the object's memory location. If the class does have this neat method and it returns a string, this string is printed. In our example, I want the string representation of a `Node` object to be its ID, and so I return `self.id` which is indeed a string.

Here's an interactive session where I test the class, using it to model part of a family tree:

```
1  >>> from binary_tree import Node
2  >>> g = Node("Grandad", None)
3  >>> d = Node("Dad", g)
4  >>> u = Node("Uncle", g)
5  >>> m = Node("Me", d)
6  >>> b = Node("Brother", d)
7  >>> d.setLeft(m)
8  >>> d.setRight(b)
9  >>> g.setLeft(d)
10 >>> g.setRight(u)
11 >>> print g
12 N1
13 >>> print b
14 N5
15 >>> print m.getParent().getData()
16 Dad
17 >>> print m.getParent().getParent().getData()
18 Grandad
19 >>> print u.getParent().getLeft().getLeft().getData()
20 Me
```

After importing the class, I instantiate five `Node` objects. Each holds a string as its data item — the first argument in each instantiation. The first node (the Grandad node `g`), is the root node, so its parent node is set to `None` (line 2). The Dad and Uncle nodes `d` and `u` both have the Grandad node as their parent, so `g` is given as the second argument in their instantiations (lines 3–4). Finally, a Me and a Brother node are created, both with the Dad node as parent (lines 5–6).

Next, I have to set the child references right inside the tree. The Me node becomes the left child of the Dad node (line 7), the Brother node becomes the right child (line 8). In lines 9 and 10, the Dad and Uncle nodes are set to be the left and right children of the Grandad node, respectively. After all this setting up, we have the situation depicted in Figure 7.14.

Continuing the test, I `print` the `g` and `b` nodes in lines 11 and 13 and get their IDs, `N1` and `N5`, respectively. This mirrors the fact that the Grandad node was created first and the Brother node last.

Next, I start to dot my way around the objects using their access methods. In line 15, I access the data string of the parent node of `m` to get `Dad` as expected: `m.getParent()` returns the parent node of `m`, and I can go on dotting with `.getData()` to get to its data string.

In line 17, I ask for the parent node of the parent node of the Me node `m` with `m.getParent().getParent()` and ask for its data string, and indeed I get `Grandad`. And finally, in line 19 I start in the Uncle node `u`, ask for its parent (which is the Grandad node), then ask for *this* node's left child (the Uncle node is the right child) and eventually ask for its data string. Which, fortunately, is `Me`. Thus, all references seem to have been correctly installed.

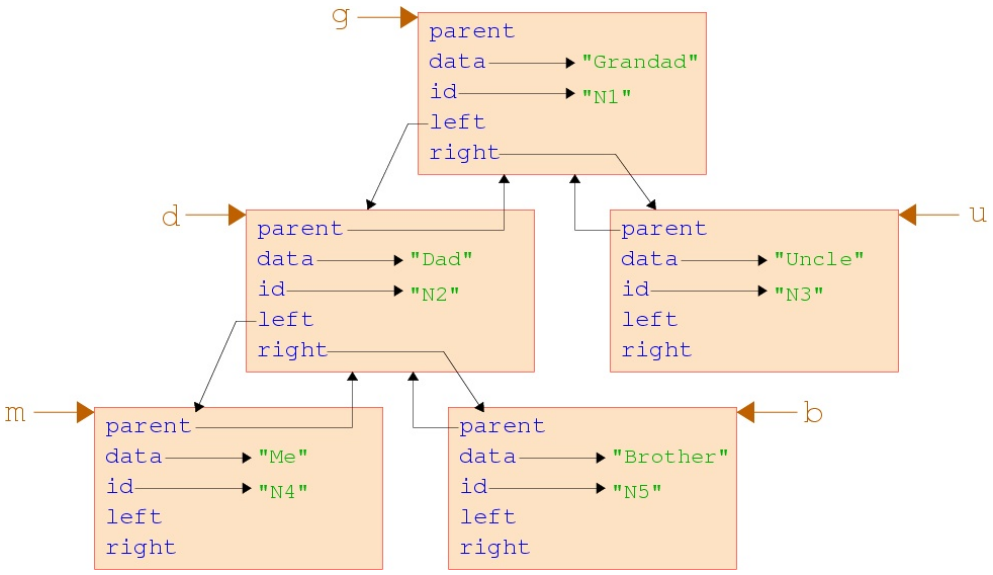


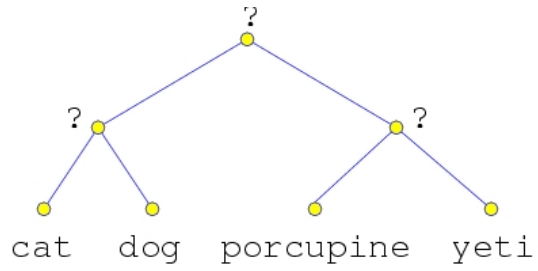
Figure 7.14: How to represent a family tree with `Node` objects. `None` references not shown.

You can imagine that I could go on creating child nodes forever. Being a recursive data type, `Node` has no restrictions as to the depth of its tree instances. And note that although the structure very quickly becomes hard to fully comprehend for a human, the picture is the same in every local part of the tree: A node has two children and a parent, and this organization can then be repeated ad libitum.

The idea of this whole tree thing was to represent a phylogeny of DNA sequences, remember? Now that we have a class for representing DNA sequences and a class for representing binary trees, let's put them together. The `Node` test you saw used strings as data items in its nodes; now we should put `Seq` objects in them instead.

Let's say that we're interested in the evolutionary history of the cat, the dog, the porcupine and the yeti. More specifically, we wish to put them together in a phylogeny. They are all extant and none of them is an ancestor of any of the others, so we'll create a phylogeny where these four species are the leaves. The question is how exactly the tree should look. The root node, their most recent common ancestor, must have two child nodes, and each should be

the parent of two of our model species. Thus, we need to pair them up, and the members of each pair should be closer related than any member of one pair with any member of the other pair. Let's just make a guess to begin with and pair cat with dog and porcupine with yeti.



The parent node of the dog and cat nodes should be the most recent common ancestor of dog and cat, and similarly the parent node of the porcupine and yeti nodes should be their most recent common ancestor. As these prehistoric common ancestor creatures are extinct, we do know not their DNA, so we're going to have to guess here, too. The animal to be represented by the parent node of these two nodes, the root, is even more extinct, so again its DNA is a mystery.

What we'll do is we'll *propose* a phylogeny with the known DNA sequences of our four model species in the leaves, and with random DNA sequences in the internal nodes, and then we'll discuss what to do with it and what we can say about it. So, let's get to work.

The program in Figure 7.15 imports the two classes and then begins by creating the `Seq` objects we'll need. In lines 4–7, `Seq` objects for our four model species are instantiated (ignoring the Latin names), and in lines 9–10 the second level node `Seq` objects are created. The `dat` object of line 9 is meant to represent the DNA sequence of the most recent common ancestor of cat and dog, and the `yetipine` object created in line 10 holds the DNA of the ancestor of porcupine and yeti. Finally, in line 12 the `Seq` object representing the DNA sequence of the root node, the most recent common ancestor of all animals in the tree, is instantiated. Let's pretend that the DNA strings inserted in the objects are the correct ones for our model species and pure speculation for the three internal nodes (don't dwell on the fact that the DNA of cats and dogs and porcupines may have changed during this chapter. Blame it on progress).

Now we have the `Seq` objects we'll use as data items in the tree nodes; next, we need to construct the tree. In line 14, we create the root node, assigning as its data item the `datipine`, the `Seq` object representing the overall most recent common ancestor. The root doesn't have a parent node, so the second argument is `None`.

In lines 16–17, we create the second layer nodes `i1` and `i2`. Again, each is given the proper `Seq` object as its data item. In lines 19–22, the four leaf nodes are created. And finally in lines 24–29, all the remaining tree connections are set: First `i1` and `i2` are installed as the left and right children of the root node, respectively, and then the `leaf1` node and the `leaf2` nodes (representing cat and dog) are installed as the children of `i1` while the `leaf3` and `leaf4` nodes (representing porcupine and yeti) are installed as the children of `i2`.

```
from binary_tree import Node
from abstract_datatype import Seq

# create DNA sequence objects of the leaf nodes:
cat = Seq("cat", "-", "CATCATCAT")
dog = Seq("dog", "-", "CGTCACCAT")
porcupine = Seq("porcupine", "-", "CTTAT")
yeti = Seq("yeti", "-", "GCAGGCG")

# create DNA sequence objects of the internal nodes:
dat = Seq("dat", "?", "CATGATCGG")
yetipine = Seq("yetipine", "?", "CTTAAA")

# create DNA sequence object of the root node:
datipine = Seq("datipine", "?", "CGCATAG")


# create root node holding datipine sequence:
root = Node( datipine, None)

# create internal nodes:
i1 = Node(dat, root)          # holds dat sequence
i2 = Node(yetipine, root)     # holds yetipine sequence

# create leaf nodes:
leaf1 = Node(cat, i1)         # holds cat sequence
leaf2 = Node(dog, i1)         # holds dog sequence
leaf3 = Node(porcupine, i2)   # holds porcupine sequence
leaf4 = Node(yeti, i2)        # holds yeti sequence

# create node connections:
root.setLeft(i1)
root.setRight(i2)
i1.setLeft(leaf1)
i1.setRight(leaf2)
i2.setLeft(leaf3)
i2.setRight(leaf4)


h = totalHamming(root)
print "Total Hamming distance:", h
```

Figure 7.15: Program `phylogeny.py`, part 1.

It may seem a lot of hassle — and indeed later we'll create trees in a much more ingenious way — but for now, let's just close our eyes and relax for 10 seconds, trusting ourselves to have connected all the objects in the right way, and then push on.

Now we have a proposed phylogeny. How good a guess is it? Naturally, there is no way of knowing since the dat, yetipine and datipine are all dead and gone, but we can still do a bit of analysis on the tree. We do have our Hamming distance. Recall that the Hamming distance was a means of estimating the evolutionary distance between two DNA sequences. The greater the Hamming distance, the more nucleotide differences between the sequences — or, in other words, the more genetic modifications would be needed to transform one into the other. Which supposedly is what has happened if one is an ancestor of the other.

With our tree, we postulate several ancestral relationships: Each branch (also called *edge*) in the tree represents an ancestral relationship where the parent node is an ancestor of the child node. Thus, a way of estimating how good a guess our tree is would be to add up all Hamming distances induced by it; i.e., to calculate all Hamming distances between neighboring nodes in the tree and add them together. That would give an idea of the total number of genetic modifications necessary to transform the root sequence into the second layer sequences and further into the four model species. That sounds all right, yes?

What we need, therefore, is a method which can cycle through a tree — *any* tree, of course, we're generalists, find all its edges, calculate all the Hamming distances and return the sum. Again, since a tree may have any size, how can we make sure we get to all its branches?

As we are using a recursive data type, of course we need a recursive function to work with it! The way to solve it is to look at it locally: Given some node in the tree, the total Hamming distance of the *subtree* rooted at this node is the sum of the distances from the node to its children PLUS the total distances of the subtrees rooted at each child. Get it? The total distance of a tree with only one node is 0. The total distance of a tree with a root and two children is the sum of the two distances from the root to each child PLUS the sum of the total distances of its subtrees (which is 0). And so forth.

So, in order to calculate the total Hamming distance of some tree rooted at a given node, first calculate the total distance of its subtrees. Then work on the actual node and its children. That's the algorithm behind the function presented in Figure 7.16.

In line 4 we check whether the current node is a leaf. If so, the total Hamming distance of the subtree which has this node as its root is 0, and 0 is returned. If the current node is *not* a leaf, it has two children. Through the variables `leftson` and `rightson`, we get a handle to these two child `Node` objects in lines 7 and 9. As part of our calculation, we need the total Hamming distances of the *subtrees rooted at these two children*, so we make recursive calls to the function in lines 11 and 12.

Next we need the Hamming distance between the current node and its two children. To keep the code as readable as possible, we obtain the `Seq` objects of the current node and its child nodes and store them in local variables `this_Seq`, `left_Seq` and `right_Seq` (lines 14–16). After that we can directly call the `distanceTo` method on `this_Seq` in lines 17 and 18. To be able to follow the progress of the recursive function calls, we print the calculated distances between this node and each of its children in lines 19–22; to get to the

```
1  def totalHamming( node ):
2      """Calculate the total Hamming distance in the
3      tree rooted at the given node"""
4      if not node.getLeft():    # node is a leaf
5          return 0
6      # left son (root of left subtree):
7      leftson = node.getLeft()
8      # right son (root of right subtree):
9      rightson = node.getRight()
10     # Total Hamming distances of the subtrees:
11     H_lefttree = totalHamming(leftson)
12     H_righttree = totalHamming(rightson)
13     # Distances from this node to its child nodes:
14     this_Seq = node.getData()
15     left_Seq = leftson.getData()
16     right_Seq = rightson.getData()
17     H_left = this_Seq.distanceTo(left_Seq)
18     H_right = this_Seq.distanceTo(right_Seq)
19     print "H(%s, %s)= %d"%(this_Seq.getName(),
20 left_Seq.getName(), H_left)
21     print "H(%s, %s)= %d"%(this_Seq.getName(),
22 right_Seq.getName(),H_right)
23     # return the sum:
24     return H_left + H_right + H_lefttree + H_righttree
25 h = totalHamming(root)
26 print "Total Hamming distance:", h
```

Figure 7.16: Program `phylogeny.py`, part 2.

species names, we have to call `getName` on the respective `Seq` objects. And finally, in line 24 the sum of it all is returned.

Note that this recursive function does not always recurse (make a recursive call to itself): When called on a leaf, it doesn't. And that's good, because otherwise the function would never terminate, right? Any recursive function should at some point *not* recurse. To calculate the total Hamming distance of a tree, then, we call the function on the root of the tree in line 25.

Figure 7.17 shows the output when the program is run. First of all, you can check that the total Hamming distance, 21, is the correct one since you get the

same number from adding all the individually calculated Hamming distances between all parent/child pairs of nodes — and note that all six pairs appear in the output. We assume our Hamming distance method works; sceptics are welcome to calculate the distance for all six pairs by hand.

```
1 H(dat, cat) = 3
2 H(dat, dog) = 5
3 H(yetipine, porcupine) = 1
4 H(yetipine, yeti) = 5
5 H(datipine, dat) = 4
6 H(datipine, yetipine) = 3
7 Total Hamming distance: 21
```

Figure 7.17: Output from program `phylogeny.py`.

Looking closer at the information printed during the recursive calls, we see that the first distance printed is that between the dat and the cat. The cat node is the left-most leaf (see Figure 7.15), and the dat node is its parent. In other words, the first node for which output is printed is the dat node, even though the first function call is made on the root (the datipine). Give it a thought if necessary, and you'll realize it's not a surprise. Look at Figure 7.16: We make the recursive calls in lines 11 and 12, *before* printing the distances in lines 19–22. Thus, before the first call on the root node can get to printing the distances to its child nodes, all recursive calls must have terminated. When a call is eventually made to the dat node, a further recursive call is made on its left son, the cat node; this call in turn terminates right away wihtout printing any output (line 5) since the cat node is a leaf. The same thing happens with the call to the other son, the dog node. These are the first calls which do not cause further recursive calls. Hence, the flow of control returns to the dat node call and proceeds to print the output, namely the local distances between the dat and the cat and dog, respectively.

Of course, the challenge now is to find the tree which *minimizes* the total Hamming distance. I leave that to you, for now at least. In any case, you've seen how one can *traverse* a recursive data structure like a tree using a recursive function. Most often, a recursive data type class will offer such recursive methods which can perform certain look-up tasks, calculations, or even modifications to the structure. We'll get to that later on.

In this final tour de force, we've used classes and objects for *modeling real-world phenomena*. This is one of the main applications of *object oriented programming* in general. If the data you are dealing with are trains, time tables and railroad track junctions, it's sometimes extremely helpful if you in your program can deal with train objects, time table objects and junction objects and have them associate to and communicate with each other much as they do in real life.

7.4 A genetic algorithm

vend tilbage til list comprehension, filter, reduce, map .. paa objekter

should be the DNA sequence which minimizes the added Hamming distances to all four of them.

model

destructor method

Kurt.

7.5 Referencing methods through the class name

When you call a method, say `doSomething`, via an object, say `myObject`, from some class, say `MyClass`, you've seen that the way to do it is through the dot operator:

```
myObject = MyClass()
myObject.doSomething()
```

This is exactly what you're doing when calling a string, list or dictionary method, as in:

```
>>> a = [1, 2, 3]
>>> a.pop()
3
```

Here, `a` is an object instantiated from the *list class* (which is called `list`), and `pop` is a method from this class. Like for any other method of any other class, the first parameter of `pop` is the object reference, `self`, and `a` is implicitly passed to `pop` as the first argument.

You've seen that the (better) way of accessing a class attribute is via the class name: e.g. with `Node.id` as in Figure 7.13. In fact, you can also access the methods of the class directly via the class name. Refering again to Figure 7.13, the expression `Node.getLeft` is perfectly legal. The question just is how the `getData` method would know from *which* node to return the data? This way, the method is not called directly on a specific `Node` object, so no object reference can be extracted from the call and given to the `self` parameter.

The answer is that the object reference must be given explicitly as the first argument. Thus, you would do as shown in line 3 here:

```
1 >>> from binary_tree import Node
2 >>> g = Node("Grandad", None)
3 >>> Node.getData(g)
4 'Grandad'
5 >>>
```

```
6 >>> g.getData()
7 'Grandad'
```

Get it? In line 3, I call the method through the class and explicitly pass the specific object which the method should work with as the first argument. In line 6, I call the method the normal way, where the object reference is passed as the hidden, first argument. The two ways are completely equivalent.

This is exactly what you're doing when calling a string, list or dictionary method through `str`, `list`, or `dict`, respectively, rather than through a specific string, list or dictionary, as in:

```
>>> a = [1, 2, 3]
>>> list.pop(a)
3
```

There, we've tied up a loose end. Now you understand why there are two ways of calling the same string/list/dictionary method.

Now you know about:

- Constructors
- The stupid but necessary `self` parameter
- Object attributes
- Class attributes
- Access methods
- Recursive data types
- The `__str__` method

Chapter 8

Regular expressions

If you're searching a string for some specific substring, you can use the string method `find` listed in Figure 3.10. But if you're searching for something not quite specific, you can't use `find`.

A **regular expression** is a description of a string **pattern**, rather than a specific string. Python's module for dealing with regular expressions is the topic of this chapter, but regular expressions are not a Python invention; the theory of regular expressions has roots in mathematics and theoretical computer science and dates back to the 1950's.

8.1 Inexact string searching

Myostatin, also known as **Growth/Differentiation Factor 8**, is a protein which limits the growth of muscle tissue. It has been found in humans, mice, cattle and other animals. If the function of Myostatin is inhibited, either due to a mutation in the associated gene or by treatment with substances that bind to the Myostatin molecules preventing them from performing their normal duties, the natural muscle growth regulation is suppressed and the animal develops muscles two to three times the normal size, as illustrated in Figure 8.1.

A gene is a part of a genome which encodes a protein. The cellular machinery decodes the gene's DNA sequence three nucleotides at a time, translating each triplet (**codon**) into an amino acid. The final protein is the result of chaining together all these amino acids.

Figure 8.1: A normal mouse compared to a mutant mouse deficient in myostatin.
From S.-J. Lee, A.C. McPherron, *Curr. Opin. Devel.* 9:604–7. 1999.

Most genes are found in several variants, called **alleles**, across a population. Often, all alleles give rise to functional proteins, but sometimes one specific allele causes a degenerate protein product, or no protein at all, and the normal protein lacks in the host organism, possibly leading to a disease.

Let's assume that the most common variant (referred to as the **wild type**) of the myostatin gene is 30 nucleotides long and has this sequence:

```
ATTCAGAATCGACTAGGATCAGCTAAGCTA
```

Further, let's assume that the gene is **polymorphic** at positions 10, 20 and 30 in humans, such that position 10 is either a C or a G, position 20 is either a C, a T or a G, and position 30 is either an A or a T. We can describe the polymorphic gene this way:

```
ATTCAGAAT(C|G)GACTAGGAT(C|T|G)AGCTAAGCT(A|T)
```

Here, the red parts indicate the polymorphisms; one and only one of the characters in each set of red parentheses is part of the gene. Thus, among humans we can expect to find up to $2 \times 3 \times 2 = 12$ different alleles of the gene.

Imagine that we have partially sequenced the genome of each of 10,000 people, some of whom seem to be myostatin deficient (i.e., they make Californian governor Arnold Schwarzenegger look like Goofy). The question is, which allele(s) of the myostatin gene can be associated with the myostatin malfunction, judging from this dataset? To answer this question, we first need to figure out 1) if the gene is present in the sequenced part of each individual's genome, and 2) if so, which allele the individual has. Thus, in all 10,000 incomplete genomes, accepting variation at positions 10, 20 and 30, we need to search for the string `ATTCAGAATCGACTAGGATCAGCTAAGCTA`. What to do?

We could, of course, call `str.find` on all combinations of the 10,000 genomes and the 12 alleles, but that sort of repetitive labor is hardly appealing. And fortunately, Python's module for regular expressions, `re`, comes in handy. Here's how to search a single genome (dummy) string for any one of the 12 alleles:

```
>>> import re
>>> p = "ATTCAGAAT(C|G)GACTAGGAT(C|T|G)AGCTAAGCT(A|T)"
>>>
>>> genome = """ACTGACATGCTGTATATCATGCATGTACGTGACTGACT
... ACGTACTGACTGATCGTAGTCTACGTAGCATCGACGTATATGATATGATC
... ACTTAGAGATTCAGAATGGACTAGGATGAGCTAAGCTACTATAGTGTAAA
... CGTACTGACTACTGACTGCATACGTACGATGTATCATGCTAGCATGA"""
>>>
>>> if re.search(p, genome):
...   print 'This genome contains the gene'
...
This genome contains the gene
```

The way we describe the pattern we're looking for (line 2) seems very intuitive. The exact way the pattern string is interpreted is this:

- Letters outside parentheses match themselves.
- The `|` symbol is special and means "Match the expression on the left or else the expression on the right".
- Parentheses are used to *group* parts of the pattern into subpatterns.

The parentheses are necessary to limit the effect of the `|` operator; the pattern `"ATTCAGAATC|GGACTAGGATC|T|GAGCTAAGCTA|T"` would match one of the strings `"ATTCAGAATC"`, `"GGACTAGGATC"`, `"GAGCTAAGCTA"` or `"T"`.

The `search` function of the `re` module is called on a pattern and a string, in that order (as in "search for *this* pattern in *this* string"). If the pattern is not found, `re.search` returns `None`. Otherwise it returns a so-called `MatchObject` which has various methods for processing the search result. If all you need to know is whether your pattern is found in the string, you can just check if `re.search` produces something which is not `None`, as is done in line 9 above.

The `re.search` function first compiles (translates) the given pattern into a special object called a `RegexObject` object. Such an object has a search method, and the string also passed to `re.search` is then given to the search method of the `RegexObject` object which returns the result. Thus, `re.search` operates in two steps; a pattern translation step and a search step.

Since we are going to repeat the myostatin gene search 10,000 times, in principle the exact same pattern translation is going to be repeated 10,000 times also.[1] This is silly — and for complex patterns it's inefficient — and fortunately there is a way to avoid it. We can precompile the pattern and then use the search method of the resulting `RegexObject` object directly, rather than using `re.search` each time. This is illustrated next. Imagine that `genome` is a list of genome strings (see Figure 8.2 below):

```
1  >>> import re
2  >>> from samplegenomes import genome
3  >>> p = "ATTCAGAAT(C|G)GACTAGGAT(C|T|G)AGCTAAGCT(A|T)"
4  >>> pcomp = re.compile(p)
5  >>> for i, g in enumerate(genome):
6  ...   if pcomp.search(g):
7  ...     print "Genome %d contains the gene"%(i+1)
8  ...
9  Genome 1 contains the gene
10 Genome 2 contains the gene
11 Genome 3 contains the gene
12 Genome 4 contains the gene
```

[1] Python *will* cache, i.e. keep, recently compiled regular expressions and reuse them, but if you're working with several patterns, it might not help.

```
13 Genome 5 contains the gene
14 Genome 6 contains the gene
```

In line 3, I compile the gene pattern `p` once and for all and assign the resulting `RegexObject` to variable `pcomp`. In line 4, I iterate through all the genomes (recall that the function `enumerate` gives me both the index and the content of each list cell). In line 5, I call the `search` method of `pcomp`; `pcomp` corresponds to the pattern `p` that I'm searching for, so calling its search method on the genome string to search in, `g`, means "look for the pattern `p` in `g`".

It seems now we know how to find out which of the 10,000 individual, partially sequenced genomes that contain the genes. To be able to discuss the possible association of each of the 12 gene alleles to the myostatin deficiency, we cut down our data set to only contain these individuals. Next, we must detect which allele each of them has; i.e., we need to know what "incarnation" of the pattern was found in each individual case.

One of the methods of a `MatchObject` object is called `group`. It is used to retrieve the substring which matched the pattern. If you call it without arguments, it returns the substring that matched the whole pattern. If you give it one or several integer arguments, you get a tuple containing the substrings that matched the parenthesized subpatterns of your overall pattern string; these are numbered left-to-right starting from 1; the substring matching the whole pattern has number 0.

Let's put it all together in a program now. Figure 8.2 is just a collection of sample genomes, put together in a list, `genomes`. The main program in Figure 8.3 imports this string as well as the `re` module.

```
1  genome = [
2      'CAGTTGAGACTCCGATATCGACTGATCAGCGCTACTGCTGGAGATTCAGAAT
3  GGACTAGGATTAGCTAAGCTTATCGGATATATCGAGTTCTCCGCGAGAGATCACG',
4      'CGATATTCAGAATCGACTAGGATTAGCTAAGCTACGACTGACTATATAAGAC
5  GCGTTGTATGTCGTGCGTACGTGTACGTGTACGATCGCTAGCTACATTCAGAATC',
6      'CGTTGATTATCGACTGTCGATTCAGAATGGACTAGGATGAGCTAAGCTATCT
7  GACTGACGTACGTACGTATTATAACGCGCGGGGACTACCGAACTGCGTCGATCAT',
8      'CATACTAGACTCATGACTACTACTAGCTACGGGCGCGGGGATATCTCCCATA
9  CTTAATTCAGAATCGACTAGGATGAGCTAAGCTTTCGATCGATCTATTATAGCGC',
10     'CGTACGATTCAGAATGGACTAGGATTAGCTAAGCTACATGCATGACGTACGT
11 ACTGACTACGTAGCTATGTATCGCGACTGCGCATGCTAGACGCCCATACACTACT',
12     'ACTAGCTACTACGATACTATTATCGATCGACGATCGCGCGCTAATATATCAC
13 TAGCTACTAGCTGAATTCAGAATCGACTAGGATCAGCTAAGCTACATACGTATCG']
```

Figure 8.2: Program `samplegenomes.py`.

In line 4 of Figure 8.3, I define the same myostatin gene pattern, `p`. In line 6, I compile it. Line 7 prints a header line with markers at the polymorphic sites. Line 8 iterates over the (indices and) genomes in the `genomes` list.

In line 10, I call the `search` method on the compiled pattern object on the current genome, `g`. I keep the result of this search in a variable, `mo`; if I get a match object, I'll need to access methods inside it in a moment. In line 12 I check whether I get a match at all, i.e. whether `mo` is `None` or a `MatchObject` object. In the latter case, I call `group` on it with no arguments in line 14 to print the complete allele matched in the current genome. And finally in line 16, I call `group` again, this time passing the arguments $1, 2, 3$ to indicate that I want the substrings that matched the three subgroups of my main pattern, i.e. `"(C|G)"`, `"(C|T|G)"` and `"(A|T)"`. The result of the call is a tuple, in this case of single-letter strings. I then pass this tuple to the string method `join`, using a `/` as the joining character. Eventually, I can put *this* string inside another string to enclose it in parentheses. The program output is shown in Figure 8.4.

```
1  from samplegenomes import genome
2  import re

3  # gene pattern to look for:
4  p = "ATTCAGAAT(C|G)GACTAGGAT(C|T|G)AGCTAAGCT(A|T)"

5  # compile pattern into SRE_Pattern object:
6  srep = re.compile(p)

7  print "            |         |         |" # SNP positions
8  for i, g in enumerate(genome):

9      # search for pattern in each genome:
10     mo = srep.search(g)

11     # did we get None or an SRE_Match object?
12     if mo:
13         # print full pattern match
14         print "%d: %s"%((i+1), mo.group()),

15         # print the three subpattern matches:
16         print "(%s)"%"/".join(mo.group(1, 2, 3))

17 print "            |         |         |"
```

Figure 8.3: Program `myostatin.py`. A "SNP" is a single-nucleotide polymorphism.

8.2 Character classes and metacharacters

Actually, the real myostatin is a protein consisting of 1125 nucleotides. If you go to the American National Center for Biotechnology Information (NCBI) web page, you can find the human myostatin in their protein database at this URL: `http://www.ncbi.nlm.nih.gov/entrez/viewer.fcgi?db=`

```
            |         |         |
 1: ATTCAGAATGGACTAGGATTAGCTAAGCTT (G/T/T)
 2: ATTCAGAATCGACTAGGATTAGCTAAGCTA (C/T/A)
 3: ATTCAGAATGGACTAGGATGAGCTAAGCTA (G/G/A)
 4: ATTCAGAATCGACTAGGATGAGCTAAGCTT (C/G/T)
 5: ATTCAGAATGGACTAGGATTAGCTAAGCTA (G/T/A)
 6: ATTCAGAATCGACTAGGATCAGCTAAGCTA (C/C/A)
            |         |         |
```

Figure 8.4: Output from program `myostatin.py`.

`protein&val=51452098`. Along with a whole lot of other information, the protein's amino acid sequence itself is also there:

```
  1 mqklqlcvyi ylfmlivagp vdlnenseqk envekeglcn actwrqntks srieaikiqi
 61 lsklrletap niskdvirql lpkapplrel idqydvqrdd ssdgsleddd yhattetiit
121 mptesdflmq vdgkpkccff kfsskiqynk vvkaqlwiyl rpvetpttvf vqilrlikpm
181 kdgtrytgir slkldmnpgt giwqsidvkt vlqnwlkqpe snlgieikal denghdlavt
241 fpgpgedgln pflevkvtdt pkrsrrdfgl dcdehstesr ccrypltvdf eafgwdwiia
301 pkrykanycs gecefvflqk yphthlvhqa nprgsagpcc tptkmspinm lyfngkeqii
361 ygkipamvvd rcgcs
```

The amino acids:
alanine (A)
arginine (R)
asparagine (N)
aspartic acid (D)
cysteine (C)
glutamic acid (E)
glutamine (Q)
glycine (G)
histidine (H)
isoleucine (I)
leucine (L)
lysine (K)
methionine (M)
phenylalanine (F)
proline (P)
serine (S)
threonine (T)
tryptophan (W)
tyrosine (Y)
valine (V)

The numbers on the left side are simply index numbers: Each line has 60 amino acids in groups of 10. In total, therefore, we see that myostatin has 375 amino acids. As each represents the decoding of a 3-nucleotide DNA codon, the myostatin gene must consist of $3 \times 375 = 1125$ nucleotides.

Say that we load this string into a Python variable. How do we get rid of the annoying numbers? And the whitespace (newlines, spaces) in between? It's clear to us exactly what we want to throw out, even though it's not one particular number or one particular kind of whitespace, but it's not clear to any string method. We could repeatedly call the string method `replace` to first throw out all newline characters, then all space characters, then all 1's, then all 2's, then all 3's, etc. ad nauseam. But surely there must be a better way; after all, we're removing stuff following a certain *pattern*..

Of course there is! With regular expressions, you can easily specify "any digit" as opposed to "this particular digit". And similarly, you can specify "any whitespace character" rather than "this particular whitespace character". Here's how:

```
>>> re.sub("\s|\d+", "", s)
```

The `re` function `sub` (short for "substitute") is similar to the string function `replace`. It takes three string arguments P, R, S and returns a copy of the string S where all occurrences of the pattern P have been replaced by the replacement string R. In this example, the replacement string is empty, and the pattern is `"\s|\d+"`. Again we see the `|` indicating that the pattern either matches `"\s"` or `"\d+"`.

In a regular expression pattern, `\s` denotes the character class consisting of all whitespace characters: Space, newline (`\n`), tab (`\t`), formfeed (`\f`), carriage return (`\r`) and vertical tab (`\v`). Thus, in a pattern, `\s` matches *one* occurrence of any whitespace character. In our example, when `re.sub` sees such a character in the string `s` it is searching, it replaces it with the empty string, since the whitespace character is one possible incarnation of the overall pattern, `"\s|\d+"`.

The other half of the pattern, `"\d+"`, also contains a character class, namely `\d`. This character class contains all the digits 0, 1, ..., 9, so `\d` matches *one* digit. There is also a `+`, though, and in a regular expression, a `+` is one of the **metacharacters**, i.e. characters which take on a different meaning than usually. It means that rather than matching one digit, the pattern `"\d+"` matches *at least* one; in other words any non-empty series of consecutive digits, such as `"1"`, `"181"` and `"361"`. Generally, a `+` in a pattern matches one or more occurrences of the expression it follows.

Of course, you can also design your own character class. You do that simply by putting the characters of the class inside a pair of square brackets; e.g., to match exactly one lowercase vowel, you can use this pattern: `"[aeiouy]"`. To match one odd digit, use `"[13579]"`. To match one of the characters A, B, C or D, use `"[ABCD]"`, or use simply `"[A-D]"`: When used inside a character class, the `-` symbol denotes a *range of consecutive characters* ("consecutive" in the natural sense in case of letters and numbers). To match any character *other than* A, B, C or D, use `"[^A-D]"`: When used as the first character inside a character class, `^` *negates* the class. If it doesn't come first, it has no special meaning and matches itself like any other character of the class.

Finally, rather than the `|` symbol, we can also use character classes to match the polymorphisms of the myostatin gene:

```
"ATTCAGAAT([CG])GACTAGGAT([CTG])AGCTAAGCT([AT])"
```

We only need the parentheses if we still want to subgroup the polymorphisms so that we are able to extract their actual manifestations; if we only want to detect the gene and don't care which allele, we just need

```
"ATTCAGAAT[CG]GACTAGGAT[CTG]AGCTAAGCT[AT]"
```

As you know from the strings chapter, the backslash, \, is used to "escape" the usual meaning of the following character. There are several very useful special escape sequences (backslash'ed letters) which correspond to certain regular expression character classes, other than the digits and the whitespace characters you've seen already. They are listed in Figure 8.5. Note in particular that if you put a period in a regular expression pattern, it means "match any character other than a newline". You'll see some of these character classes brought to use later in this chapter.

Character class	Description
`\d`	The class of digits: `"[0-9]"`
`\D`	The class of non-digits: `"[^0-9]"`
`\s`	The class of whitespace characters: `"[\n\f\r\t\v ]"`
`\S`	The class of non-whitespace characters: `"[^\n\f\r\t\v ]"`
`\w`	The class of alphanumeric characters: `"[a-zA-Z0-9_]"`
`\W`	The class of non-alphanumeric characters: `"[^a-zA-Z0-9_]"`
`.`	The class of all characters except newline.

Figure 8.5: The special character classes of Python's regular expressions.

Besides +, there are also the two metacharacters `?` and `*`. Where + means matching "at least one occurrence of the preceding expression", `?` means "zero or one occurrence" and `*` means "any number of occurrences", respectively. E.g., to search a string for a number which may or may not begin with a minus sign and may or may not contain a decimal point, you could do like this:

```
>>> a = "Let's use 3.14 for pi.."
>>> b = ".. and -273 for the absolute zero temperature"
>>> pattern = "-?\d+(\.\d+)?"
>>> re.search(pattern, a).group()
'3.14'
>>> re.search(pattern, b).group()
'-273'
```

Regular expressions quickly start to look very complicated, but that's the price you pay for their expressiveness and power. The pattern defined here, `"-?\d+(\.\d+)?"`, first matches zero or one minus signs and then a series of at least one digit. These digits should be followed by something (the subpattern in parentheses) which must occur 0 or 1 times, as dictated by the `?` in the end. The job of the subpattern is to match a decimal point and then at least one digit. However, the `.` symbol is a special symbol matching any non-newline character (cf. Figure 8.5); to eliminate this extraordinary behavior, we have to

escape the `.` by putting a backslash in front of it. Thus, `"."` matches any non-newline character, `"\."` matches a period/decimal point. Hence, `(\.\d+)?` matches either a decimal point and some digits, or nothing.[2]

As you saw earlier, a `-` in a character class indicates a range of characters, as in `"[A-D]"` matching either the letter A, B, C or D. But when used in a pattern *outside* a character class, `-` just means `-`. The same is true for the `^` symbol: When put first inside a character class, it negates the class. When otherwise used in a pattern, it means "match the beginning of the string"; see Figure 8.7. This subtlety of discriminate meanings inside and outside character classes may be challenging.

With `*`, `+` and `?`, you match an unspecific number of sequential occurrences of the preceding expression. You can also match a *specific* number of occurrences by putting the number in curly braces. Danish phone numbers have 8 digits; the following example shows how to match a valid Danish phone number enclosed in parentheses:

```
1 >>> s = "Jakob (86197491): jakobf@birc.au.dk"
2 >>>
3 >>> re.search("\(\d{8}\)", s).group()
4 '(86197491)'
5 >>>
6 >>> re.search("\((\d{8})\)", s).group(1)
7 '86197491'
```

Here you see two slightly different patterns both finding the phone number. The central part of both patterns is `"\d{8}"`; this expression matches a repetition of 8 digits. To match the parentheses around the phone number, we need the parenthesis symbols, but again, these symbols already have a special meaning in a regular expression pattern (namely subgrouping). Thus, we need to *escape* them in order to simply recognize good old parentheses. Therefore, the pattern in line 3 begins with `"\("` and ends with `"\)"`.

When looking for a Danish phone number in a text containing parenthesized phone numbers, it's not enough to just look for 8 digits. Such a search might return the first 8 digits of a 10-digit phone number, and that's why we need to match the parentheses as well. However, we really only want to extract the phone number, not the parentheses, so we need to identify the phone number part as its own subgroup. That's what's done in the pattern in line 6; it hasn't exactly become more readable, but give it a good scrutiny and you'll see that it makes sense. To retrieve the matched subgroup (i.e., the phone number), I pass the argument 1 to the `group` method of the `MatchObject` object resulting from the search; `group()` would still give me the whole thing.

If you are looking for not only Danish 8-digit phone numbers but any phone numbers with, say, 8, 9 or 10 digits, you can change your pattern slightly:

[2] The search/group combo statement `re.search(p, s).group()` results in an error if the pattern `p` is not found in the string `s`. In general therefore, be sure that it's there, or better yet, check if `search` returns `None` before calling `group`.

`"\d{8, 10}"` matches a series of digits of length 8, 9 or 10. Thus, the curly braces may also be used to indicate a *range* of repetitions. To summarize the metacharacters for repetition, I've put them all in Figure 8.6.

Normally, `*`, `+` and `?` match *as many* characters as possible. Observe:

```
>>> s = "I like ABBA, AHA and DAD"
>>> re.search("A.*A", s).group()
'ABBA, AHA and DA'
```

Thus, the first time a substring is found which matches the pattern `"A.*A"` (an A followed by anything followed by another A), this substring is extended as much as possible while still matching the pattern. For this reason, `*`, `+` and `?` are called **greedy** operators. If one instead wants to match as *few* characters as possible, one should just put a `?` after the operator. Thus, to get only `"ABBA"`, go with the pattern `"A.*?A"`:

```
>>> re.search("A.*?A", s).group()
'ABBA'
```

Note that using the non-greedy versions of the operators will not *skip* a longer match to get to a shorter one further down the string; it just chooses the shorter one if there are several ways to constitute the first match according to the pattern.

Symbol	Description
`+`	Match *one or more* repeated occurrences of the preceding pattern, and as *many* as possible.
`*`	Match *zero or more* repeated occurrences of the preceding pattern, and as *many* as possible.
`?`	Match *zero or one* occurrence of the preceding pattern, and *one* if possible.
`{n}`	Match `n` repeated occurrences of the preceding pattern.
`{a, b}`	Match *n* repeated occurrences of the preceding pattern with `a<=n<=b`.
`+?`	Match *one or more* repeated occurrences of the preceding pattern, and as *few* as possible.
`*?`	Match *zero or more* repeated occurrences of the preceding pattern, and as *few* as possible.
`??`	Match *zero or one* occurrence of the preceding pattern, and *zero* if possible.

Figure 8.6: Repetition characters of Python's regular expressions.

Characters	Description
.	The character class of all characters except newline.
*	Any number of occurrences of the preceding pattern.
+	One or more occurrences of the preceding pattern.
?	Zero or one occurrence of the preceding pattern.
{ }	For repeating the preceding pattern the given number of times.
[ ]	Defines a character class which matches one of the characters in the class.
( )	Defines a subgroup.
|	Matches either the pattern to its left or the pattern to its right, delimited by the enclosing subgroup if one exists.
\	Used for predefined character classes like \d, and to escape the special characters of this table.
^	When used as the first character inside a character class, it negates the class. Otherwise, it matches the beginning of the search string (see Figure 8.8).
$	Matches the end of a line, or the position between the last character and a newline (if any) at the end of the search string (see Figure 8.8).

Figure 8.7: All the characters with special meaning in Python's regular expressions. To match the actual character, you have to escape it by preceding it with a backslash. E.g., to match a |, use the pattern "\|". NB: Inside character classes, all except the last three match themselves.

Look at Figure 8.7 listing all the characters with special meaning in Python's regular expressions. In fact, all but the last three (\, ^ and $), *lose* this special meaning and are taken literally if they appear inside a character class. Here is an example illustrating how the meaning of . changes depending on whether it appears inside or outside a character class:

```
1 >>> if re.search("[.]+", "Anything"):
2 ...   print '"[.]+" matches'
3 ... elif re.search(".+", "Anything"):
4 ...   print '".+" matches'
5 ...
6 ".+" matches
```

The pattern in line 1 has no match in the search string "Anything" because it contains no dots. The pattern in line 3, however, *does* have a match, since here, the . means "any character". Thus, to match one of the special characters themselves, either put it in a singleton character class, like [*] or

`[{]` (or even `[[]` to match a `[`), or backslash it as suggested in the caption of Figure 8.7.

One of the exceptions is the metacharacter `^`. When put *first* in a character class, it negates the class; in other positions, it matches itself. Outside a character class, it matches the beginning of the search string, i.e. a *position* of zero width. Here's an example:

```
>>> if re.search("^flex", "lexical flexibility"):
...     print "match!"
... else:
...     print "no match"
...
no match
```

Even though the string `"lexical flexibility"` definitely contains the substring `"flex"`, the search returns no result because the pattern begins with a `^`, and the search string does not start with `"flex"`.

The metacharacter `$` has the opposite meaning: it matches the end of the search string — again, a position of zero width, not a character. Thus, to match a pattern to a search string only if the pattern is in fact the entire search string, you can use both `^` and `$` as shown here:

```
>>> numbers = ["86197491", "8002529141"]
>>> pcomp = re.compile("^\d{8}$")
>>> for number in numbers:
...   if pcomp.search(number):
...     print number, 'is a Danish number'
...
86197491 is a Danish number
```

Even though the pattern simply matches 8 consecutive digits, it doesn't match the second search string containing a 10-digit phone number, because the 8 digits both have to start at the beginning and end at the end of the search string. In other words, only search strings which are strings of 8 digits and contain nothing else are matched.

The last exception to the "is taken literally inside a character class" rule is the backslash, `\`. I'll return to this painful issue in Section 8.4.

Finally, note that when defining a character class, you may combine whichever characters you need with predefined character class metacharacters. For example, you could match either `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `+`, `-`, `*` or `/` with the character class `[\d+\-*/]`. Note, that you have to backslash the minus symbol to avoid it being interpreted as a range indication, as it usually is inside a character class.

8.3 Flags, functions and methods of the re module

When you compile a regular expression pattern, you can tell Python to interpret the pattern in several special ways, e.g. to ignore case, by setting a suitable **flag** when calling `compile`; you can think of a flag as a signal. The way to set a flag is to include it as the second argument to `compile`, after the pattern string. The flags are constants defined inside the `re` module, so you dot your name to a flag via `re`. The most important flags are listed in Figure 8.8. For instance, to search for our old friend the myostatin gene while ignoring the case of the nucleotides in the search string, we could set the `IGNORECASE` flag like this:

```
>>> s = 'gcatattcagaatggactaggatgagctaagcttaat'
>>> p = "ATTCAGAAT(C|G)GACTAGGAT(C|T|G)AGCTAAGCT(A|T)"
>>> pcomp = re.compile(p, re.IGNORECASE)
>>> pcomp.search(s).group()
'attcagaatggactaggatgagctaagctt'
```

Flag	Meaning
DOTALL	Make . match any character including newlines.
IGNORECASE	Do case-insensitive matches.
MULTILINE	Multi-line matching which affects ^ and $.

Figure 8.8: Regular expression compilation flags. Access them via `re`, as in `re.DOTALL`.

The other two flags only take effect when you're dealing with search strings stretching over several lines. Normally, the `.` metacharacter matches any character except newline, and you'd use the expression `".*"` to match anything within a single line. If you want to match anything at all, including newline characters, you set the flag `re.DOTALL` when compiling your pattern.

When you set the `MULTILINE` flag, you change the behavior of the `^` and `$` metacharacters. Usually, they only match the very beginning and end of the search string, respectively, but if the `MULTILINE` flag is set, they accept beginnings and ends of *lines* within the search string as well. Thus in multiline mode, `^` matches the beginning of the search string as well as the position between any newline and the first character after the newline within the search string, and similarly, `$` matches the end of the search string as well as the position between a newline and the character immediately preceding it.

You have seen three of the functions in the `re` module: `compile`, `search` and `sub`. Figure 8.9 shows the most useful ones, and I'll discuss a few of them. Except for `compile`, all functions can be called either via the module, as in

```
re.search(p, s)
```

where the pattern is the first argument, or via the `RegexObject` resulting from a precompiled pattern, as in

```
p_comp = re.compile(p)
p_comp.search(s)
```

The `findall` function returns a list of *all*, non-overlapping matches of a pattern in a search string; e.g. all words beginning with *a* and ending with *e*:

```
>>> s = """America believes in education: the average
professor earns more money in a year than a
professional athlete earns in a whole week. (E. Esar)"""
>>> re.findall("a\w+e", s)
['average', 'athlete']
```

The string function `str.split` splits some string at each occurrence of some specific substring. With regular expressions you don't have to be specific: You can use `re.split` to split your string at every occurrence of some *pattern*. Say that you want the subsentences of some sentence, splitting it at every occurrence of a punctuation mark. Proceed like this:

```
>>> s = """If a man is offered a fact which goes
against his instincts, he will scrutinize it closely,
and unless the evidence is overwhelming, he will
refuse to believe it. If, on the other hand, he is
offered something which affords a reason for acting in
accordance to his instincts, he will accept it even on
the slightest evidence. The origin of myths is
explained in this way.   (Bertrand Russell)"""
>>> for subsentence in re.split("[.,;!?]", s):
...   print subsentence
...
If a man is offered a fact which goes against his
instincts
 he will scrutinize it closely
 and unless the evidence is overwhelming
 he will refuse to believe it
 If
 on the other hand
 he is offered something which affords a reason for
acting in accordance to his instincts
 he will accept it even on the slightest evidence
 The origin of myths is explained in this way
   (Bertrand Russell)
```

Method name and **example**	**Description**

`re.compile(p[, flags])`
 	
`>>> pcomp = re.compile("A.+B", re.DOTALL|re.MULTILINE)`
	Compile the pattern `p` into a `RegexObject` which provides the methods listed in the rest of this table. The optional `flags` argument is a series of flags (cf. Figure 8.8), separated with `|`.

`re.split(p, s[, maxsplit)`
 	
`>>> re.split("\d", "a1b2c3")`
`['a', 'b', 'c', '']`
	Split the string `s` at each occurrence of the pattern `p`. If optional argument `maxsplit` is given, perform at most this many splits. Return a list of substrings. If the pattern is enclosed in parentheses, the substrings matching the splitting pattern are included in the list.

`re.sub(p, r, s[, count])`
 	
`>>> re.sub("\d", "*", "a1b2c3")`
`'a*b*c*'`
	Return a copy of the string `s` in which all occurrences of the pattern `p` have been replaced by the string `r`. If optional argument `count` is given, do at most this many replacements.

`re.subn(p, r, s[, count])`
 	
`>>> re.subn("\d", "*", "a1b2c3")`
`('a*b*c*', 3)`
	Same as `sub` but returns both the string and the number of replacements performed in a tuple.

`re.match(p, s[, flags])`
 	
`>>> re.match("b1", "b1c2").group()`
`'b1'`
	`re` module version: Return `MatchObject` if pattern `p` is found *at the beginning* of the string `s`; otherwise return `None`. Optional argument `flags` is interpreted as with `compile`.

`pcomp.match(s[, pos[, endpos]])`
 	
`>>> pcomp = re.compile("b2")`
`>>> pcomp.match("a1b2c3", 2, 4).group()`
`'b2'`
	`RegexObject` version: Return `MatchObject` if precompiled pattern `pcomp` is found *at the beginning* of the string `s`; otherwise return `None`. If optional arguments `pos` and `endpos` are given, limit the search to the string `s[pos:endpos]`.

`re.search(p, s[, flags])`
 	
`>>> re.search("b2", "a1b2c3").group()`
`'b2'`
	`re` module version: Return `MatchObject` if pattern `p` is found in the string `s`; otherwise return `None`. Optional argument `flags` is interpreted as with `compile`.

`pcomp.search(s[, pos[, endpos]])`
 	
`>>> pcomp = re.compile("b2")`
`>>> pcomp.search("a1b2c3", 0, 4).group()`
`'b2'`
	`RegexObject` version: Return `MatchObject` if precompiled pattern `pcomp` is found in the string `s`; otherwise return `None`. If optional arguments `pos` and `endpos` are given, limit the search to the string `s[pos:endpos]`.

`re.findall(p, s[, flags])`
 	
`>>> re.findall("\w(\d)\w(\d)", "a1b2c3d4")`
`[('1', '2'), ('3', '4')]`
	`re` module version: Return list of non-overlapping matches of the pattern `p` in the string `s`. If `p` contains a subgroup, return a list of subgroup matches; if it has n groups, the list will be a list of n-tuples. Optional argument `flags` is interpreted as with `compile`.

`pcomp.findall(s[, pos[, endpos]])`
 	
`>>> pcomp = "\w\d\w"`
`>>> pcomp.findall("a1b2c3d4")`
`['a1b', 'c3d']`
	`RegexObject` version: Works as the module version but with the usual use of optional arguments `pos` and `endpos`.

Figure 8.9: `re` functions/methods, called either via `re` or via a precompiled pattern `pcomp`.

Continuing this example, let's say that you also want the actual punctuation marks found, i.e. the substrings matching the splitting pattern. In this case, you should simply enclose the pattern in parentheses; that gives you a list of both substrings and splitting patterns. I.e., assuming that no two punctuation marks appear right after each other, you can get just the punctuation marks by slicing out every other item of the list returned by `findall` when given the pattern `"([.,;!?])"`, starting at index 1:

```
>>> re.split("([.,;!?])", s)[1::2]
[',', ',', ',', '.', ',', ',', ',', '.']
```

Most of the functions and methods described until now have all been accessible either through the `re` module itself or through a compiled `RegexObject`. The one exception is `group` which you call on a `MatchObject` resulting from a successful regular expression search. There are a few other useful methods available through a `MatchObject`; Figure 8.10 tells their story.

Method	Description
`group()`	Return the string matched by the pattern. If the pattern contains subgroups, you can access the string matched by each subgroup by passing the rank of the subgroup to `group`: `group(1)` returns the first subgroup, `group(1, 3, 4)` returns a tuple containing the first, third and fourth subgroups, etc.
`start()`	Return the starting index of the match.
`end()`	Return the ending index of the match.
`span()`	Return a tuple containing the (start, end) indices of the match.

Figure 8.10: `MatchObject` methods.

8.4 Raw strings, word matching, backreferencing

Before we move on, I have to introduce you to the concept of **raw strings**. The thing is that when Python reads a string in your program, it silently performs a transcription of what you've typed into the actual value it keeps in its memory, called a **string literal**. This is really neither very complicated nor very new, in fact, since what I'm basically talking about here is the way the special characters like newline and tab are treated. If your code contains the string

```
"Friends\nfrom\nabroad"
```

Python creates the string literal

```
Friends
from
abroad
```

from it. Thus, what are actually *two* characters in your string, `\n`, are converted into *one* character, a newline. This mechanism is called escaping: The backslash indicates that the next character should not be taken literally but rather its special meaning should be applied.

When we're dealing with regular expression patterns, again we are using the backslash to invoke a special meaning for certain characters, namely those used for the predefined character classes: `\d`, `\D`, `\s`, `\S`, `\w` and `\W`, as listed in Figure 8.5. In this section, I'll introduce two new ways in which the backslash can invoke a new, special meaning for some characters; the first is for recognizing word boundaries, the second is used for something called backreferencing, and I'll return to that a little later.

Python defines a word as a substring composed of only alphanumeric characters, so recognizing a word amounts to recognizing a substring delimited by anything *not* an alphanumeric character, or by whitespace. To match such a zero-width word boundary in a regular expression, you use the metacharacter `\b`. But here, trouble arises: `\b` is used in *ordinary* strings as the backspace character. Look:

```
>>> print "Calm\bzone"
Calzone
```

Thus, when Python sees the string `"Calm\bzone"`, it first transcribes it into the literal `"Calzone"`, converting the `\b` into a backspace. If you intended to use the string as a regular expression and compile it as such, you'll lose the regular expression meaning of `\b`, namely word boundary matching.

The solution to this intricate problem is to tell Python *not* to do its initial transcription, i.e. the step which converts the *two* characters `\b` into *one* backspace. You do that by putting an `r` in front of the string, like this:

```
r"Calm\bzone"
```

Any string preceded by `r` is left untranscribed — and is called a "raw" string.

In this book, all program examples are printed in nice colors: Keywords are blue, strings are orange, comments are red, etc. This coloring is done by a Python program which takes as its input another Python program and scans it for keywords, strings etc., creating an output with colors which I can then import in the word processing system used for the book. Figure 8.11 shows a simple version of this coloring program where I only color the keywords.

First, I import our old friend, the `inred` function from Figure 5.2; I'll use it to print all keywords in red. In line 3, I define all the keywords reserved in Python. And in line 11, I input a filename of some other Python program from the user.

The plan is to build one regular expression pattern for recognizing any keyword. The obvious solution would be something like:

```
"and|as|assert|break| ..  |yield"
```

However, there is a problem with this pattern: It matches the keyword substrings *anywhere* in the source code, i.e. also when contained inside a longer word; not just as separate words. Thus, I need to match a word boundary on either side of each keyword as well. Hence, I arrive at this pattern:

```
r"\band\b|\bas\b|\bassert\b|\bbreak\b| ..  |\byield\b"
```

I have to make it a raw string; if I don't, see what my pattern degenerates into:

```
>>> p = "\band\b|\bas\b|\bassert\b|\bbreak\b|\bclass\b"
>>> print p
anaasserbreaclass
```

In other words, forgetting the `r` in front of a regular expression pattern containing the word boundary metacharacter \b is disastrous; you end up matching the (strangely wonderful) substring `"anaasserbreaclass"` rather than either `"and"` or `"as"` or `"assert"` or `"break"` or `"class"`!

To build the pattern, then, I should join all the keywords with the substring `"\b|\b"` in between, and then add `"\b"` to the front and `"\b"` to the end. That's what I do in the mysterious line 15.

Next, I compile the pattern and initialize a counter variable, `count`, to keep track of the number of keywords found and replaced.

In line 18, I open the file given by the name that the user provided.[3] In line 19, I enter a loop in which I iterate through the lines of the file one by one: Recall that `for line in file` fetches one line at a time into memory, whereas `for line in file.readlines()` would fetch the whole file at once. For large file sizes, that would be inefficient or even unfeasible.

In line 20, I use the `findall` method of my compiled pattern object, `pcomp`, to get a list of all occurrences of keywords in the current line. Next, I wish to replace each keyword `w` by `inred(w)`. That's done in lines 21–23.

In line 21, I enter a loop going through the individual keywords from the list. For the string replacing, I use the `re` function `subn` which returns the new string as well as the number of replacements performed. To match and replace the current word, `w`, again I need to match word boundaries as well, so the pattern I'm replacing is `"\b%s\b"%w`, and the replacement string is `inred(w)`. The returned tuple is assigned to `line` and `n` before the next word of the found keywords is replaced. Note that the variable keeping the current line

[3] For any real application, you should use the `try/except` construct when dealing with files since you do not control the file system and things may go wrong.

```
1  from stringcolor import inred
2  import re

3  keywords = ['and', 'as', 'assert', 'break',
4              'class', 'continue', 'def', 'del',
5              'elif', 'else', 'except', 'exec',
6              'finally', 'for', 'from', 'global',
7              'if', 'import', 'in', 'is', 'lambda',
8              'not', 'or', 'pass', 'print', 'raise',
9              'return', 'try', 'while', 'with',
10             'yield']

11 filename = raw_input("Enter name of python program: ")

12 # match any substring which is a word and which is
13 # identical to one of the keywords; i.e.  build the
14 # pattern "\band\b|\bas\b|\bassert ..":

15 pattern = r"\b%s\b"%r"\b|\b".join(keywords)

16 pcomp = re.compile(pattern)
17 count = 0

18 f = open(filename)
19 for line in f:

20     words = pcomp.findall(line)

21     for w in words:
22         line, n = re.subn(r"\b%s\b"%w, inred(w), line)
23         count += n

24     # skip trailing newline, keep indentation:
25     print line.rstrip()

26 f.close()

27 print inred("\n%d keywords found"%count)
```

Figure 8.11: Program `pythonkeywords.py`.

of code, `line`, is repeatedly reassigned to point to an updated line with a new keyword replaced. In line 23, I update the total count of replacements.

Outside this loop, in line 25, I print the processed line. Since reading a file line by line means that the last character of each loop element, `line`, is a newline, simply `print`'ing it would give me two newlines: The one already in the string, and the one produced by `print`. So I remove the trailing newline before printing, using string method `rstrip`. I don't use `strip` which removes all whitespace at either end of a string, because it is important to keep

the leading indentation of a line of Python code.

Eventually in lines 26–27 I can close the file and print the overall replacement count. Figure 8.12 shows a trial run where I feed the program to itself. Note that, e.g., the substring `line` in

```
for line in f
```

is *not* written as

```
line
```

because of the word boundary matching using `\b`. Note also, however, the undesirable coloring of keywords in the comment lines — and, for that matter, the keyword strings in the list which really shouldn't be colored either. I'll leave the details of this possibly quite tricky improvement to the reader.

Two final remarks on matching word boundaries: (1) Inside a character class, `\b` still represents backspace. And (2) There is also a metacharacter `\B`. It means the opposite of `\b`, namely "match any position which is *not* a word boundary".

To round off the discussion of raw strings, I'll now bring to life the terrors of matching the backslash character itself with a regular expression. The backslash *does* have a life on its own, you know; it is not only the chaperone of other special characters. Let's play around a little, circling the problem.

```
>>> s = "This string contains a backslash: \"
  File "<stdin>", line 1
    s = "This string contains a backslash: \"
                                           ^
SyntaxError: EOL while scanning single-quoted string
```

Oops, we haven't even started building our pattern yet, and already we get an error? "End of line scanning a single-quoted string"?? The reason is that a Python string may not end in a (single) backslash; Python will try to escape the following character, namely the string delimiting quote or ping symbol, and then it erroneously thinks the strings doesn't end anywhere. Alas, you have to backslash your backslash. If it appears somewhere inside a string (and doesn't precede some character which may be escaped, like *n* or `t`), it's okay.

```
Enter name of python program:  pythonkeywords.py
from stringcolor import inred
import re

keywords = ['and', 'as', 'assert', 'break',
            'class', 'continue', 'def', 'del',
            'elif', 'else', 'except', 'exec',
            'finally', 'for', 'from', 'global',
            'if', 'import', 'in', 'is', 'lambda',
            'not', 'or', 'pass', 'print', 'raise',
            'return', 'try', 'while', 'with',
            'yield']

filename = raw_input("Enter name of python program:  ")

# match any substring which is a word and which is
# identical to one of the keywords; i.e.  build the
# pattern "\band\b|\bas\b|\bassert ..":

pattern = r"\b%s\b"%r"\b|\b".join(keywords)

pcomp = re.compile(pattern)
count = 0

f = open(filename)
for line in f:

    words = pcomp.findall(line)

    for w in words:
        line, n = re.subn(r"\b%s\b"%w, inred(w), line)
        count += n

    # skip trailing newline, keep indentation:
    print line.rstrip()

f.close()

print inred("\n%d keywords found"%count)

43 keywords found
```

Figure 8.12: Output from program `pythonkeywords.py` when run on itself.

```
>>> s = "This string contains a backslash: \\"
>>> t = "Here's another: \ right in the middle"
>>> re.search("\\", s)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "/usr/lib/python2.4/sre.py", line 134, in search
    return _compile(pattern, flags).search(string)
  File "/usr/lib/python2.4/sre.py", line 227, in _compile
    raise error, v # invalid expression
sre_constants.error: bogus escape (end of line)
```

Now what's all this, then? "Bogus escape (end of line)"?! Well, as explained, Python first transcribes our string into a string literal: The two-character string \ \ becomes the one-character string \ — which is what we wanted, right? We wanted to match a single backslash. But when interpreting this string literal as a regular expression, again Python sees the poor, misunderstood backslash as an escape symbol associated with the next character — which is the final quote of the pattern string. Thus, inside the regular expression, we once more need to backslash the backslash. What we need, therefore, is that the string that reaches the regular expression compiler is: `"\\"`. In total, therefore, to match *one* backslash in a regular expression, the pattern must contain *four* backslashes: `"\\\\"`! ☹

— Unless of course you use a raw string. A raw string containing two backslashes will continue to contain two backslashes; when it reaches the regular expression compiler, it still contains two backslashes, and at this point then, the first acts to escape the usually special meaning of the second, namely that of escaping the following character, and they are reduced to just match one, single backslash character. Confused? I don't blame you.

```
>>> p = re.compile(r"\\")
>>> print p.search(s).group()
\
>>> print p.search(t).group()
\
```

Let's move on to backreferences. A backreference in a pattern allows you to specify that the exact substring matching a subgroup defined earlier must also be found at the current location in the search string. A backreference is written as a backslashed number; the number refers to the rank of the subgroup you want (following the system also used by the `group` function). If your pattern contains two subgroups, then `"\2"` refers to whatever substring matched the second subgroup. Unfortunately, for ordinary strings, the "backslash-

number" notation is already taken,[4] and so you're bound to get the same trouble as with `\b` unless you use raw strings.

Backreferencing is especially useful when you do string replacements where the replacement string contains the actual substring that matched the pattern. Consider the keyword coloring task again; the job is to print all Python keywords in red. I.e. replace each keyword by the *same* keyword with some magic stuff before and after. Figure 8.13 shows a much more elegant solution.

```
1  # more elegant solution than pythonkeywords.py
2  from stringcolor import inred
3  import re
4  from keyword import kwlist
5  filename = raw_input("Enter name of python program: ")
6  pattern = r"(\b%s\b)"%r"\b|\b".join(kwlist)
7  pcomp = re.compile(pattern)
8  f = open(filename)
9  s, count = pcomp.subn(inred(r"\1"), f.read())
10 f.close()
11 print s
12 print inred("\n%d keywords found"%count)
```

Figure 8.13: Program `pythonkeywords2.py`.

In line 4, I simply import a list of keywords, `kwlist`, from the `keyword` module, so I don't have to type them in manually. The next lines are the same as in Figure 8.11, except now I subgroup my entire regular expression pattern (line 6): This way, the actual keyword matched by the pattern can be backreferenced as group 1. I then compile the pattern and assign the compiled object to variable `pcomp`, and then I open the file. This time, I don't go over the file line by line; to keep the program short, I read the entire file at once with `f.read()` in line 9 and pass it as the search string to the `pcomp` method `subn`.

The replacement string pattern I pass to `subn` is then simply `r"\1"`, or rather `inred(r"\1")`. The `"\1"` refers the first group of `pcomp`, i.e. the actual, specific keyword matched. In other words: When you find any keyword from the list, pass it to the `inred` function. One line is all it takes. Nice, eh? As usual, `subn` returns the replacement string as well as the number of replacements made, and so I can print both in lines 11 and 12. Figure 8.14 shows the output when running the program on itself.

There are more to regular expressions than what you've seen in this chapter, though I think I've shown you the essentials. The only way to get the hang of regular expressions is to play around with them, so do that! To learn

[4] In an ordinary string, a backslashed number n inserts the character with the ASCII code n.

```
Enter name of python program:  pythonkeywords2.py
# more elegant solution than pythonkeywords.py

from stringcolor import inred
import re
from keyword import kwlist

filename = raw_input("Enter name of python program:  ")

pattern = r"(\b%s\b)"%r"\b|\b".join(kwlist)
pcomp = re.compile(pattern)

f = open(filename)
s, count = pcomp.subn(inred(r"\1"), f.read())
f.close()

print s
print inred("\n%d keywords found"%count)

7 keywords found
```

Figure 8.14: Output from program `pythonkeywords2.py` when run on itself.

more, consult the `re` module's pages on the Python web site at `http://www.python.org/doc/current/lib/module-re.html`, or read this tutorial: `http://www.amk.ca/python/howto/regex/`.

Now you know about:

- Regular expressions and the `re` module
- Metacharacters
- Character classes
- Raw strings
- Backreferencing

Index