Introductory Programming in Python

March 17, 2013

Chapter 1

Basic concepts

1.1 What is a Program?

Simply put, a program is a set of instructions on **how** to take some **input** data and produce **output** data. Programs can be written in many languages, such as PERL, Python, C, Ruby, Pascal, etc... Even a stylised form of natural languages such as English, known as pseudo code, can be used to describe a program effectively.

Input -> Program -> Output

It is important to realise that the **program** determines what input is satisfactory, and what input will cause error. Input cannot be used to change how a program works. Similarly, the program defines what output will be produced. If you need a program to handle input differently, you need to write a new program. Likewise, if the output isn't what you need, you will need to change the program too.

Ultimately a program, no matter what language it is written in, consists of a some atomic actions/instructions, each one an instruction that cannot be divided into a sequence of 'smaller' or 'less complex' instructions. These instructions are composed (as in mathematical composition) in various manners, usually by issuing them in sequential order, but also by defining the result of one instruction as the operand of another.

Instructions (or sets of composed instructions) can be divided into two groups, those that produce a result, and those that don't. For example, adding two numbers, 3 and 4, together has a result (7). More specifically, it has a **value**, i.e. some data the program can continue to work with. Contrast this with an instruction that puts a line of text on the screen. It produces no value as a result. Instructions which produce a value are called **expressions**, instructions which produce no value are called **statements**

In a similar manner to statements, expressions can be combined with other expressions, to form complex expressions. Expressions are combined using **operators** such as the mathematical functions sum, difference, product, quotient, etc... Thus 1+1 is also an expression that has **a single value** despite the fact that it is a composite of multiple sub-expressions.

Which we can compare to the much simpler version of the same thing in python

```
a = 12
b = 12
if a*b == 144:
    a = 1
```

The basic point is to illustrate that programs are made up of small, well defined, **easy to understand** steps in sequence. As in chess, where each individual piece can only move in a very limited number of ways, yielding a tiny number of potential moves per piece per turn, these moves can be combined in a near infinite number of ways and often grouped together into common techniques or strategies. So too can the simple statements of a programming language be combined in infinite ways to produce complex but meaningful results.

1.2 Everyone can program!

Believe it or not, you've programmed before. You've programmed your friends, and do so every time you give them directions. After all what is a set of directions but a sequence of instructions.

```
OUT OF Cape Town TAKE the N1
TAKE the Sable rd. Offramp
AT the fork VEER LEFT
AT the traffic lights TURN LEFT
AT the NEXT traffic lights TURN RIGHT
AT the traffic circle TURN LEFT
AT the NEXT traffic circle TURN RIGHT
AT the t-junction TURN LEFT
```

You will note that some (in fact a hell of a lot) of the words in the directions to my place are capitalised. In the language of giving directions, these are pretty much our most basic instructions. That is, they are the **atomic** instructions of the program. This means that each of these instructions cannot be divided into a combination of more basic instructions For example, LEFT, on it's own doesn't make sense. Likewise, TURN alone is vague. Thus TURN LEFT would be considered an atomic instruction, which along with other atomic instructions could be used to build up a more complex program of instructions.

The portion of the directions left in lowercase are labels or names for things that are not common to all sets of directions, most often places specific to the set of directions being given. These have values, and would be our expressions.

Examining the directions we have

OUT OF meaning I must first be **in** a named place before performing the next instruction

TAKE meaning to drive along, or turn off onto a named offramp

AT continue until the named place or situation is reached **before** performing the next instruction

VEER meaning to stay in a particular lane as the road splits

TURN LEFT, TURN RIGHT self explanatory

NEXT meaning the next object of specified type encountered

At first the description of these concepts may seem obvious, but recall that computers, the machines executing your programming instructions, have an IQ of 0. They are not intelligent, fiendishly annoying at times perhaps, but never intelligent, never aware, never capable of the massive amounts of understanding and contextualization done by the human brain in order to draw logical conclusions. They are designed and built to understand and execute only specific very basic steps. So what is implicit in the instructions contained in a set of directions given to us, must be explicitly defined for a computer.

1.3 Data representation and translation of real world problems

Now that the concept of sequences of statements has been thoroughly flogged to death, to what do these statements apply? They apply to **data**! But data in a computer, like instructions, must be simple and well defined, or at the very least able to be broken down into multiple well defined simple pieces. In general computers work only with numbers. The pictures you see on screen, the text you are reading, the sound you hear when playing MP3s are all numbers. The actual physical devices attached to the computer are what are responsible for transforming the numbers with which the Central Processing Unit, or 'brain' of a computer, deals into humanly recognisable phenomena such as sound waves and images. Until the screen, or the speakers, are reached, everything is numbers. So it stands to reason that the most basic, atomic, unit of data in a computer is a number. Fortunately, modern programming languages are capable of dealing with numbers and sequences of numbers in a few different ways. Integers and Reals can be considered atomic data units in almost every modern computer language, as can text in the form of a string of characters in sequence.

As programs are usually written to solve problems occurring in the real world, it falls to the programmer to translate the problem being solved into something the computer can deal with, i.e. numbers. This is a bit like those annoying word problems we encountered in junior school mathematics.

> Jane has seven apples, Mary has four, Bob has one. They pool their resources, and divide the apples equally. How many apples does each one receive?

The most difficult concept to grasp when learning to program is the ability to translate a problem expressed in words into a set of instructions that describe the solution to the problem. Learning a programming language doesn't teach one to program, it merely provides one with a specific set of tools with which one can solve a problem. Learning how to apply these tools is the true skill to programming, and this comes primarily with experience. The problem set out above is ridiculously simple, and you've already worked out the answer in your head, but **how did you do it?** Describe the process! But what if there were 1000 people involved, and many thousands of apples. Working it out in your head becomes a tedious task, but the basic process you followed in your head for three people applies equally well to the case of a thousand people. And so for our first exercise in programming let us translate the word

problem into the atomic (most basic) statements and atomic data units that can be used to provide us with the answer. Assume we are provided with only the following statements and expressions to work with, and that statements are numbered from 1 upwards in the order in which they appear in our program:

- EXPRESSION – `input()`: get a number from input
- STATEMENT – `labelname = #`: Assign the value of number to a label for storage
- STATEMENT – `labelname += #`: Addition of the second number to the number stored in labelname
- STATEMENT – `if # != # {}`: check if the two numbers are not equal. If they are not equal perform any instructions within the braces `{}`
- STATEMENT – `GOTO #`: Instead of executing the next statement, execute the statement numbered `#`
- STATEMENT – `labelname /= #`: Division of the number stored in labelname by `#`
- STATEMENT – `print(#)`: Output of a number to the screen

Note that `#` can be either an actual number, the name of a label storing a number, or the instruction `input()` which is the number received as input.

> Each of a number of people has at least one apple. They pool their resources and divide the apples equally. For any given number of people and the number of apples each of these people has, how many apples will each person receive?

Given only the above statements and expressions to work with, there are some important questions that need answering

- Do we need to know who started with how many apples?
- How do we represent how many apples there are?
- How can one determine the total number of people?

```
1: apples = 0
2: people = 0
3: a = input()
4: people += 1
5: if a != 0 {
6:     apples += a
7:     GOTO 3
   }
8: apples /= people
9: print(apples)
```

1.4 Exercises

1. What is a program?

2. What is the difference between an expression and a statement?

Chapter 2

Invocation

2.1 Starting the interactive shell

Starting Python is easy.

1. We will be using IDLE to get things started
2. On the Desktop you will see a link named "Python"
3. To open the Interpreter click on **"Run"** > **"Python Shell"**
4. The screen will open up and look similar to the following

```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012,
   01:25:11)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more
   information.
>>> WARNING: The version of Tcl/Tk (8.5.9) in use may be
    unstable.
Visit http://www.python.org/download/mac/tcltk/ for
   current information.
>>>
```

What you see now is known as the Python interactive shell. The first line tells you what version of Python you are running. The second line gives you some information about how this particular copy of Python was built, and on what system it is running. The third line lists some commands you can use to get more information. Whilst in the interactive shell, you can enter Python expressions and see their results immediately. Try it now, type in a simple mathematical expression such as `4 + 7`.

```
>>> 4 + 7
11
>>>
```

As you can see, the answer or result of the expression is printed out, and you are returned to the prompt »>. Now try something different: type in `a = 2 + 7`. Then, on the next line, type just `a`.

```
>>> a = 2 + 7
>>> a
9
>>>
```

This should be somewhat familiar from the end of the basic concepts section, where we were assigning a value to a name. In this case the name is `a` and the value is the result of `2 + 7`. Note that no value is printed out immediately after the assignment. The interactive shell always prints out the value of expressions, but not of statements. This means however that the labels to which we assign values, more correctly called **variables**, are expressions that evaluate to a value, which is why it printed out `9` when we input `a`.

2.2 Running a saved Python Script

Whilst ideal as a calculator and for exploring new ideas quickly, it would be pretty tedious if we had to re-enter our program into the interactive shell every time we wanted to run it. So instead we can, and in fact most often will, save our program code to a file. Program source code is plain raw text. As such word processors like Microsoft Word, Wordperfect, Open Office etc... are not suitable to the task. We will look for Idle which is specifically geared towards creating Python programs, and comes with Python.

So now, instead of running the interactive shell, let's write our first Python program, this is also known as a script. Open up the fist window again and do the following

1. Click on **File** > **Save**
2. Navigate to /home/*username*/Desktop
3. Save the file as **hello.py**
4. Type the following:

```
print "Hello World!"
```

Hit F5, or Click **Run** > **Run Module** to run your program.

```
>>>
Hello World!
>>>
```

This is a very simple program, but it should help you get the idea. The computer is only following the instructions you provided to it. Lets convert the test we did on the Interpreter to a program. Save it as **second.py**.

```
#My second Python program
4 + 7 #This should add two numbers and output the result
```

```
a = 2 + 7
a
```

Notice the first and second lines. They contain what are called **comments**. Any text following a hash character on a line in a Python program is a comment, including the hash itself. Comments are completely ignored by Python, and are there purely to annotate code and make things easier for humans reading the code. We use comments to place small reminders within the code for ourselves, or explain the logic behind particularly tricky sections. As we progress through the course, you will find the code examples used are sprinkled more and more liberally with comments explaining how they work.

Now it would be reasonable to expect that if we ran this program we would get the same results as given to us by the interactive shell. So, let's try it.

```
>>>
>>>
```

No output? Nothing? The Python interpreter (python), when called without a file name following, starts up in the interactive shell. Only then will it output the results of expressions entered. When invoked with a file name following, and that file contains Python code, Python will only produce output if explicitly told to do so. So lets use a trick from our first program. Modify the code so it looks like the following:

```
#My second Python program
4 + 7 #This should add two numbers and output the result
a = 2 + 7
print a
>>>
9
>>>
```

Ah, that's better. Again, you should recognise the print statement from the end of the basic concepts section. The print statement will be explained in greater detail the next section.

One more important consideration is that Python is **case sensitive**. A variable `a` and another variable `A` are not the same, as illustrated below ...

```
#This program illustrates how Python is case sensitive

a = 3 # assign a the value 3
A = "Hi" # assign A the value "Hi"

# Check whether assigning to A has changed a
print a

# Check that A and a are still different
print "A = ", A, " and  a = ", a
```

Running this produces the output:

```
3
A = Hi   and   a = 3
```

2.3 Exercises

1. Start the Python interpreter
2. Try using the interpreter as a calculator
3. Write the Hello World program.
4. Write the second program.
5. Write a program that prints out your name.
6. Consider the following lines of code:

```
a = 9
b = 3
a/b
```

   (a) For each of these three lines, which are expressions and which are statements?

   (b) What will the output be if these lines are entered into the python interactive interpreter?

   (c) What will the output be if I run these lines from a file/script?

   (d) What changes need to be made to produce output when I run these lines from a file/script?

Chapter 3

Basic output

3.1 The print() Statement

The most basic statement in Python is `print()`. The `print()` statement causes whatever is between the brackets to be outputted to the screen. We've already encountered it previously, now it's time to understand how it works. Start up the python interactive shell, and let's explore. Type the following:

```
>>> print(1)
1
>>> print(173+92)
265
>>> print(173+92.0)
265.0
>>> print("hello")
hello
>>>
```

As one can see, the `print()` statement outputs the **value** of the expression immediately following it to the screen, and moves to the next line. Note that the third print statement produces slightly different output, namely the extra '.0'. This is because 92.0 and 92 are different to a computer. 92 is an integer, whilst 92.0 is a real number, or in computing terms a **floating point number** or **float** for short. The differences will be covered later.

Also of importance is the expression "hello" (note the double quotes). The value of "hello" is *hello*, and this is what is outputted to the screen by the `print()` statement. *hello* is designated as a **string** by enclosing it in quotes. Try `print(hello)` without the quotes and see what happens.

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

```
>>>
```

What's going on? Welcome to your first bug! We will soon learn to dissect and understand what all that means, but for the moment it is sufficient to understand that something has gone wrong. But what? Recall from basic concepts we were able to store values in *labels* or *variables*. Python consists of a limited set of key words that have special meaning. These key words form the list of basic (atomic) statements and expressions that python knows how to handle. Whenever python encounters a word is doesn't recognise, it treats this as a label name. It obviously doesn't recognise 'hello' as a statement, and thus treats it as a **variable**. Variables must have a value, because variables are expressions in and of themselves. But we haven't told Python what the value of hello is, hence it complains `'hello' is not defined`.

The `print()` statement is not so plain and boring as it seems. It can do a few more things that are worth mentioning. Try entering `print("Jane has", 7, "apples.")`

```
>>> print("Jane has", 7, "apples")
Jane has 7 apples
>>>
```

Of course we could just as easily use `print("Jane has 7 apples")` and get the same result. However, separating the number 7 out illustrates two important things about the `print()` statement. Firstly, we can in fact output the values of any number of expressions in a comma separated list, and secondly the outputs of each of the expressions in the comma separated list are separated by a single space each.

Finally, you will notice that the `print()` statement always ends off the line, and starts a new one. Simply leaving a comma on the end prevents this, e.g. `print("Enter your name:",)`. In summary:

- The `print()` statement prints out the *values* between the brackets and then prints a new line
- An empty textttprint() statement ends the current line and starts a new one.
- The `print()` statement can print multiple expressions if they are given in a *comma separated* list. In this case, a space is included between each expression's outputted value.
- The automatic newline outputted by textttprint() can be avoided by putting a trailing comma in the expression list. This will print a trailing space instead.
- If a textttprint() statement with a trailing comma is followed by an *empty* textttprint() statement, the trailing space is suppressed.

3.2 Some String Basics

Python treats all text in units called **strings**. A **string** is formed by enclosing some text in quotes. Double quotes, or single quotes may be used. There is a small limitation to this however, being that a string cannot be broken across multiple lines.

```
'this is a string of text'
"this is also a string of text"
'this string will
    cause an error, because it spans multiple lines'
```

Trying to enter the third string into the interactive shell yields

```
>>> 'this string will
    File "<stdin>", line 1
            'this string will
                             ^
SyntaxError: EOL while scanning single-quoted string
>>>
```

EOL meaning End Of Line. If we want to introduce line breaks into strings we can use two methods. The first, and simplest, is to use *triple quotes*, meaning three double or three single quotes to indicate both the beginning and the end of the string, as in

```
>>> """this string will not
...  cause an error
...    just because it is split over three lines"""
'this string will not\n  cause an error\n    just
   because it is split over three lines'
>>>
```

The immediately obvious disadvantage is that everything between the triple quotes is taken as-is, meaning the second and third lines of my string which I indented to line up with the beginning of the first line are indented in the string itself in the form of three spaces after those \n thingies. Speaking of which, what the hell are those things? Why does our string contain stuff we didn't put there? Well let's try to print the string out ...

```
>>> print("""this string will not
...    cause an error
...    just because it is split over three lines""")
this string will not
   cause an error
   just because it is split over three lines
>>>
```

Well the \ns are gone, but what were they? Strings are sequences of characters and are one dimensional. They have no **implicit** way to specify a line break, or relative position, or which characters are where relative to which other characters in the string in two dimensions, as displayed on a screen. Hence we get the second method of specifying line breaks within a string. There are special characters known as **escape characters** which mean special things inside strings. They all start with a backslash \ which escapes the following character from the string, or in layman's terms means the following character in the string is not a 'normal' character and should be treated specially. Some important escape characters are

\n line break or New line (n from the n in new line)

\t tab

\\ a plain backslash

You will see that because a backslash already has a special meaning, namely "treat the next character specially", we can't simply put a backslash into our string. So we escape the backslash with a second backslash, meaning we actually want a backslash and not a special character.

Finally, how do we actually put quotes inside a string since they indicate the **end** of a string. The easiest solution is to mix your quotes. If you want a single quote in a string, define the string with double quotes, e.g. `"I've got this escape thing all figured out!"`. Alternatively, you can actually escape quotes within strings, to give them the special meaning that they don't end the string, e.g. `'I\'ve got this escaped thing totally figured out!'`

3.3 Exercises

1. What does the `print()` statement do generally?
2. Start the Python interactive interpreter:
   (a) Output your first name.
   (b) Output your surname.
   (c) Output your first name followed by a space followed by your surname.
   (d) Create a variable called `firstname` and put your first name into it.
   (e) Create a variable called `surname` and put your surname into it.
   (f) Using only the variables you have created, print your first name and surname again, making sure there is exactly one space between your two names.

   Quit the Python interactive interpreter.
3. Which special rules does the print statement adhere to regarding trailing spaces and newlines?
4. Consider the following code...

```
print("MUCH madness is divinest sense,")
print("To a discerning eye;")
print("Much sense the starkest madness.")
print("'T is the majority","In this, as all,
   prevails")
print("Assent, and you are sane;")
print("Demur,-you're straightway dangerous",)
print()
print("And handled with a chain.")
```

(a) What output, exactly, does the above code produce? Indicate spaces with underscores.

5. Write a program that prints "Hello".
6. Write a program that outputs a favourite piece of poetry or other prose, over multiple lines.
7. How can you print a value stored in the variable `x`?
8. How can you print the values of multiple expressions on one line?
9. Write a program that outputs your name, age, and height in metres in the following format. Make sure age is an integer, and height is a float, and not simply part of your string.
   `My name is James, I am 30 years old and 1.78 metres tall.`
10. Explain three possible ways to print a string containing an apostrophe, for example the string
    `The cat's mat`.

Chapter 4

Basic Input

4.1 Literal Values

So far our programs have been pretty uninteresting. They consistently produce the same results because they consistently act on the same data, or input, which we have specified by *hard coding* values for our inputs in the form of **literals**. A constant or literal is sort of the opposite to a variable, in that its value does not change over the course of program execution, and more importantly, its value is specified *literally*, within the program code.

As mentioned before, Python understands a fairly limited set of key words and symbols as basic statements or operators. Previously, we said that Python treats unrecognised words (used is a very loose sense) as variable names, but this is not strictly true. Python treats unrecognised words as expressions, and attempts to determine their value. Now, the word can specify a value directly, in which case it called a literal value, or simply *literal* for short.

```
#An example differentiating literals and variables
a = 3                  #a is a variable name, 3 is an
   integer literal
b = 'b'                #b is a variable name, 'b' is a
   string literal
c = True               #c is a variable name, True is a
   boolean literal
if 3*'b' == a*b:       #3 and 'b' are literals, a and b
   are variables
    print(c)           #is c literal or variable?
else:
    print "False"     #is "False" a literal or a variable
       ?

#Are False and "False" the same?
```

4.2 The input() Function

It's not very helpful to us if we must change our programs every time we want to change the data they work with. So instead we want to be able to tell our program to get input from somewhere. The simplest place to get input from is the keyboard, in the form of text entered by the user. Python provides a **function** to do just this, called `input()`:

```
>>> name = input("What is your name? ")
What is your name? James
>>> print(name)
James
>>>
```

We've seen *input* before in the basic concepts section, and there really isn't much to it. Just a few things to note. Firstly, input is a function. A function is a defined collection of statements that produce or **return** a value when executed. You could think of them as a way of turning a small set of statements into an expression. We will learn how to define our own functions later on in the course, but Python provides quite a few basic built in ones which we are going to be using before that, so let's look briefly at how they work.

- `input` is the name of the function. Like variables, functions have names to identify them. Like a variable, a function name is a label that points to the collection of statements to be executed.
- To execute the statements in a function, better known as **calling a function** we write the function's name followed by round brackets. `input()`
- Using the name just by itself treats the function like a variable.
- `"What is your name? "` is a parameter to the function. Functions are like mini programs. They always do the same thing, but they can do the same thing to different data. We put the data we want a function to operate on in between the brackets when we call it. Expressions and variables **passed** to functions in this way are temporarily known as **parameters**. Multiple parameters can be passed to a function by separating them with commas, and in general any valid expression is a valid parameter; e.g.
  `minimum(6, 2, 3+4, 4**2, 8.3, 3*1)`
  has six parameters, some of which are not just simple expressions like plain numbers.
- Despite the fact that any valid expression is a valid parameter, functions are usually quite picky about the parameters they can work with. For example, we couldn't find them minimum of a collection of strings. That only really works with integers and floats. Functions usually specify the *number* of parameters they accept. Often functions may accept less parameters than they specifically ask for, by substituting default values for the parameters not provided.

Back to `input`! `input()` captures a single line of text from the keyboard, as entered by the user. It returns a string containing the captured text, leaving out the `\n` produced when the user hits the Enter key. In addition, it takes one optional parameter, which is displayed as

a prompt prior to accepting input from keyboard. If this parameter is left out, the default prompt is an empty string, so nothing is printed.

4.3 Exercises

1. If there is a function named *my_function*, how do I call it?
2. Are functions expressions or statements? What about when they are called?
3. Start the python interactive interpreter:
   (a) Assign what the user types to a variable called `s`.
   (b) Print the value of the variable `s`.
   (c) Print `s`.
   (d) Use `input()` to prompt the user for a number, get that number, and assign it to a variable called `n`.
   (e) Print double the value of the variable `n`.

   Exit the Python interactive interpreter.
4. Write a program that asks the user for their name and stores it in a variable. Then output "Hello", followed by the user's name
5. Write a program that asks the user to enter two numbers, and prints the sum of those two numbers.
6. Write a program that asks the user to enter two numbers, and prints the difference of those two numbers.
7. Write a program that asks the user to enter two numbers, and prints the product of those two numbers. Are you sensing a pattern here?
8. Write a program that asks the user for some text, and a number. The program prints out the text a number of times equal to the number entered, without line breaks or spaces in between each repetition.

Chapter 5

Program state

5.1 The Concept of State

As we know, programs execute statements in sequential order. They flow through your code, left to right, top to bottom. As statements are executed, things happen: output is issued, calculations are done, and the **state** of your program changes. But what is state? Loosely put, state refers to all the data your program is dealing with and their **current** values at the point in time when a particular statement is about to be executed. Since the entire point of a program is to transform input data into some meaningful output data, it stands to reason that the program's state will change as it executes statements. We need a way to record the state of our program in manageable units. We shall call these units **variables**, since their values may vary as our program executes.

Variables each have a name, a type, and a value. A name is simply what we choose to call a particular piece of data. For example, we previously called the bit of data representing the total number of apples held by everyone 'apples'. We stored a number in 'apples', so this would be its type. The value of 'apples' started at 0. As more people were added to the problem, the number of apples increased. Accordingly, the value of our variable 'apples' changed.

Type refers to the type of value a variable can have. Text is not the same as a number, thus they are considered different types. One could not for example add "apples" to the number 7.

Python has five basic types:

int Integers, e.g. 1, 179835646, -3, 0

string Text strings, e.g. 'Alice', "The cat sat on the mat"

float Real numbers, e.g. 0.0, 48747.23501, -0.5

bool Boolean conditions, e.g. True, False

None None is used to specify a variable which has no type or actual value.

5.2 Assignment Statements

To create a new variable, we simply **assign** it a value, by giving it a name and setting the name equal to the value. We do this by using the **assignment statement** which takes the form `variable_name = expression`

```
>>> name = "James"
>>>
```

Here we have created a new variable, whose name is `name`, and which has the value *James*. Note, we did not have to say that `name` was of type string. When assigning a value to a variable, Python automatically sets the type of variable being assigned to the type of the value being assigned. If we go on to assign name again, this time with a different value

```
>>> name = "Jimbo"
>>>
```

the value of `name` will be changed. If we want to see what the value of a variable is we can simply print it:

```
>>> print(name)
Jimbo
>>>
```

James has disappeared. As far as the program is concerned James was never there. There is no James! Assigning a new value to a variable will obliterate the previous value of that variable. If you wish to keep the old value of a variable you need to save a copy, in a different variable.

```
>>> i = 1
>>> j = i
>>> i = 2
>>> print("i =", i, "and j =", j)
i = 2 and j = 1
>>>
```

Let's look at what we've done, step by step.

1. `i = 1`
   We create a new variable, called `i`, assign it a value of *1*, and Python sets it's type to `int`
2. `j = i`
   We create another new variable, this time called `j`, assign it a value of ... well what exactly? `i` is not a string because it's not in quotes, and it's not a number, so it's being treated as a variable. But what is `i`? Variables are implicitly considered to be their current values. So `i` is implicitly considered to be 1 (it's current value)
3. `i = 2`
   We assign a new value to `i`. The old value of `i` is discarded.

4. `print("i =", i, "and j =", j)`
   We output the current values of `i` and `j` respectively.

Step 2 highlights a number of important points about assignment. The assignment statement does something very specific. It changes the current value of the variable being assigned to the value of the expression to the right of the equals. This means that the value of the expression is determined before actually changing the value of the variable. As you can see from the output produced by our little program, assignments do not behave in the same way as a similar statement in maths does. `j = i` does not mean that `j` and `i` are always equivalent. The only thing that can be certainly said about the relationship between `j` and `i` is that they will be equal directly after the assignment statement has been executed. Both before and after that point in time, all bets are off. An assignment is **not** a description of a relationship between two expressions.
It effects a **once off** change in the value of one variable.

Consider the assignment statement `x = x + 1`. For those of you with inner mathematicians that are screaming loudly to get out, prepare yourself! What we have is an assignment to a variable (`x`) the value of an expression (`x + 1`). Let us suppose that currently the value of `x` is 1. This assignment occurs as follows.

1. `x = x + 1`
   The value of the expression `x + 1` is determined. Since `x` is 1, we can transform the expression into `1 + 1` since variables in expressions are considered to have their current value.
2. `x = 1 + 1`
   This is obviously 2, hence the value of the expression overall is 2.
3. `x = 2`
   2 is assigned as the **new** value of `x`.

5.3 Integers, Floats and Arithmetic Expressions

Obviously we want to be able do more than simply assign values to variables. We need to be able to manipulate and combine them in various ways. For integers and floats this leads directly to arithmetic notation, and its representations in Python. In general, basic arithmetic expressions can be formed in much the same way as one would express them mathematically. If `x` and `y` are either integers or floats then

- `x + y` yields Addition
- `x - y` yields Subtraction
- `x * y` yields Multiplication
- `x / y` yields Division
- `x % y` yields Modulo (or Remainder)
- `x ** y` yields Exponentiation (or raising to the power of)

The above list describes various **operators** that can be used to form arithmetic expressions. But, suppose we have the expression `2 + 3 * 4`. Which of the addition or the multiplication operators is applied first? One might think the operators are applied simply left to right, but as in maths, we have a well defined order of **precedence** specifying the order in which operators are applied. In general Python arithmetic operates exactly as it would in normal mathematics.

We know that variables have a type, and so does the value of expressions. The type of the value of an expression is generally determined by the types comprising the expression. For example, adding one integer to another in an expression always yields an integer, so the expression's type is integer. But what happens when you mix two, or more, types in an expression? In general, all participating sub-expressions have their type promoted to the most general type. By general, we mean *most capable of representing all types involved.* For example, the value 0.5 cannot be represented by an integer, but the value 3 can be represented by a float as 3.0. Hence adding a float to an integer means the integer gets promoted to a float and the type of the expression's value as a whole will be a float.

Python division allows us to do a number of things. We can divide two integers (say 5/2) and the result will be represented as a float (2.5). However, we often actually want the integer part of a division only (that is, we want the decimal part dropped), or the remainder. Python provides us with an operator to get both the integer division value and the remainder. Using % is called getting the modulo of two numbers:

```
>>> print(5 % 2)
1
>>> print("5 divided by 2 is ", 5 // 2, "remainder", 5 %
    2)
5 divided by 2 is 2 remainder 1
>>>
```

5.4 Strings and common operators

Strings can be manipulated too, but the operations we perform on them are fundamentally different. Assuming `s1` and `s2` are strings, and `i` and `j` are integers

- `s1 + s2` yields Concatenation
  This joining of two strings in the order they are listed as operands to from a single string; e.g. `"a" + "b"` yields `"ab"`
- `s1 * i` yields Repetition
  The creation of a new string repeated i times; e.g. `"a" * 3` yields `"aaa"`
- `s1[i]` yields Subscription (or indexing)
  The extraction of a single character at a specific position in the string, where the first character in the string is a position 0, the second and position 1, and so on; e.g. `"abc"[1]` yields `"b"`. i may be any expression evaluating to an integer. Negative numbers may be used, where `-i` means the i'th last position.

- `s1[i:j:k]` yields Slicing
  The extraction of a series of characters from a string starting at the character in the i'th position and continuing until the character just prior to the j'th position, extracting only every k'th character, where positions are numbered from 0, and negative numbers mean distance from end of string. Any of i, j, or k may be omitted, in which case i is treated as 0, j as the string length, and k as 1. A k value of 1 means all characters in the specified range are extracted. If j is omitted, the extraction is from the i'th position to the end of the string. Examples:
  `"The quick brown fox"[4:]` yields `"quick brown fox"`
  `"The quick brown fox"[-4:]` yields `" fox"`
  `"The quick brown fox"[:4]` yields `"The "`
  `"The quick brown fox"[4:9]` yields `"quick"`.
  `"Thequickbrownfox"[::2]` yields `"Teqikbonfx"`
  `"The quick brown fox"[4:15:3]` yields `"qcbw"`.

- `s1[:]` yields Slicing of the entire string, essentially making a complete copy of the string.

5.5 Formalisation of the Concepts of Statements and Expressions

To recap:

1. Programs consist of sequences of statements
2. Statements perform actions but do not have value
3. Statements may act on expressions
4. Expressions have a value, and when used in statements, their value is determined and substituted in place of the expression
5. Expressions may be simple expressions such as numbers or variables whose values can be determined directly
6. Expressions may be composed of simpler expressions combined using appropriate operators

5.6 Exercises

1. What does the assignment statement do?
2. What are the five basic variable types in Python?
3. If the integer variable `i` has the value 7, then what is the value of the expression `7/4`. Why?
4. How does one calculate the remainder (modulo) of an integer when divided by another number?

5. Write a program that asks the user for a number from 1 to 31. Assume the first day of the month is a Sunday. Output the name of the weekday of the day of month the user entered.

6. Write a program that outputs the letter "x" 1000 times on a single line, without intervening spaces.

7. If `s` is a string variable with the value `"Harry's Hippie Hoedown"`, then what is the value of
   `s + ":  tickets only $5"`

8. If `s` is a string variable with the value `"Harry's Hippie Hoedown"`, then what is the value of
   `s + ":  tickets only $" + "5"*3`

9. What is the value of `"ABBA was a Swedish band popular during the 80's"[0:4]`?

10. What is the value of `"ABBA was a Swedish band popular during the 80's"[-15:-7]`?

11. If the string variable `s` has the value `"ABBA was a Swedish band popular during the 80s"`, then what is the value of
    `"BAAB"+s[4:11]+"Danish"+s[18:24]+"un"+s[24:-4]+"90's"`

12. What is your favourite sport? Write a program that inputs the score units for this sport (e.g. for rugby: number of tries, conversions, penalties, time played) and works out some statistics (e.g. the total score, average points per minute, time remaining, expected final score).

Chapter 6

Conditionals

Up to now, our programs have been pretty straightforward. The computer has simply executed each statement, in order, one after the other. There has been no variation, no change in behaviour, no awesomeness. In life, we are able to change our behaviour based on our circumstances, i.e. we don't do **exactly** the same thing every day, day after day. Here are some everyday situations you might find yourself in, and some of the actions you might take as a result:

- If it's raining, or snowing for that matter, you won't go out, not without an umbrella at least.
- If it's dark and your lights are off, you'll turn them on, provided you're not trying to sleep.
- If you're hungry, and you have food at your disposal, or you have money to buy food, you'll eat something.

We are capable of using our reasoning and intuition, when deciding what to do. Computers however don't have intuition, and the only reasoning they have is the reasoning **you** provide them with. When analysing a situation we are capable of taking various complicated aspects into consideration. For example, when you are invited to a party you need to decide:

- How badly do I want to?
- How many people we be there?
- How wet will I get?
- How much trouble will I get into?

Computers on the other hand are not capable of this sort of reasoning. Whilst we can weigh up different pros and cons, are able to bend our own rules and are even capable of remaining unsure, a computer can only decide either yes or no. There are no in between feelings, doubts or possibilities for changing its mind. The logic computers use to make their decisions is called boolean logic. For a computer, something can only have two states, **True** or **False**. A computer will ask these questions instead:

- Do I want to?

- Will there be more than 5 people there?
- Will I get wet?
- Will I get into trouble?

A computer can analyse certain conditions, determine if they are true or false, and then act accordingly. There is no room for leeway, no randomness, the computer is always sure that what it's doing is correct. How does it know that it's correct? Because you told it it was.

In addition to computers only being aware of two conditions, programming languages generally have very few ways of determining what those conditions are and acting upon them. Lets have a look at that second example, "Will there be more than 5 people there":

```
>>> numOfPeople=7
>>> numOfPeople>5
True
>>> numOfPeople=-3
>>> numOfPeople>5
False
>>> numOfPeople=5
>>> numOfPeople>5
False
>>>
```

In the above example, we set the `numOfPeople` first to 7, then to -3 then to 5. See how the expression `numOfPeople>5` only returns `True` when the value stored in the expression is **more** than 5, and returns `False` otherwise.

`True` and `False` are special words to Python, known as **reserved words**, meaning they have been reserved for a particular purpose. A reserved word we have already encountered is `if`. Reserved words in Python are:

```
and       del       from      not       with
as        elif      global    or        yield
assert    else      if        pass
break     except    import    raise
class     exec      in        return
continue  finally   is        try
def       for       lambda    while
```

Note that when forming our proposition we used a specific operator, the greater than sign ">". When forming propositions we can use a specific set of operators, called **comparison operators**. If both `a` and `b` are expressions of any type:

- `a == b` proposes Equivalence.
  Is `a` equal to `b`? **Note the double equals**. In Python `=` means assignment, and `==` means is one value equal to another?.
- `a != b` proposes Non-equivalence.
  Is `a` not equal to `b`?

- `a < b` proposes Less Than.
  Is `a` less than `b`?
- `a > b` proposes Greater Than.
  Is `a` greater than `b`?
- `a <= b` proposes Less Than OR Equal To.
  Is `a` less than or equal to `b`?
- `a >= b` proposes Greater Than OR Equal To.
  Is `a` greater than or equal to `b`?
- `a in b` proposes that the **string** `a` is a substring of, i.e. exists **inside**, the **string** `b`. This can also be used in lists as you will see in a later chapter.

Propositions, being expressions themselves, follow the usual rules for composition of expressions. This means that we can have an expression, `(x > 3) == (y < 4)`, in which we have two propositions `x > 3` and `y < 4` which have been composed into the total expression by using the third proposition which proposes that both sides have the same value. The entire proposition is true only if both the sub-propositions have the same truth value at the time the expression is evaluated.

So back to our program to decide if we want to go to the party, we don't want to go to lame parties with too few people. So we only go to the party if there are more than 5 people there, i.e. if `numOfPeople>5`. But what if we don't like crowded parties either? We could evaluate if `numOfPeople<40`, so as to avoid parties with more than 40 people. But if only there were a way to combine them, so that we only go to parties that have a couple people **AND** are not too crowded. Say hello to **logical operators**, also known as **boolean operators** (operators which apply to boolean expressions). There are only three logical operators, listed below. If `expr1` and `expr2` are both propositions:

- `expr1 and expr2` is True if both `expr1` **and** `expr2` are True when the proposition is evaluated.
- `expr1 or expr2` is True if either `expr1` **or** `expr2` are True when the proposition is evaluated.
- `not expr1` is True if `expr1` is False, and vice versa.

You can use boolean operators to string together multiple statements. e.g. to decide whether or not to go to a party you might use the following conditions

```
numOfPeople>5 and numOfPeople<40 and (day=='Friday' or day=='Saturday')
```

This will produce `True` if the party has between 5 and 40 people, and it is either a Friday or a Saturday. **Note the brackets around the parts next to the or**. Brackets are used here to define which expression to evaluate first. This is because the evaluation of boolean algebra also follows a hierarchy system, *BODMASCNAO*. Your standard BODMAS operations on numbers and variables come first. Then comes your conditions (>, ==, `not` then `and` and finally comes `or`. So if we left out the brackets and only had:
`numOfPeople>5 and numOfPeople<40 and day=='Friday' or day=='Saturday'`

This could be interpreted as yes if the party has between 5 and 40 people and its a Friday, or if its a Saturday (In which case the number of people doesn't matter). Hence its very important to get your BODMAS right else you're going to be going to lame parties with only 2 other people, just because its a Saturday. Lets look at some examples:

```
>>> 3>4 and 25==20
False
>>> a=2==1 or 4+2>=6 and  not 5%2==0
>>> a
True
>>> b=True and 1==1 and (3!=3 or 14-2>5 and 5==2*2)
>>> b
False
>>> 'ell' in "Hello There"
True
>>> b or a and 4$<$=3
False
>>> a or b or 2!=2
True
>>>
```

Note how in that example we stored the result of a boolean expression to a variable (variables a and b), and when we called them later they acted just like any other boolean expression.

So now we know how to determine from a set of conditions whether or not something is true. We know whether or not to go to the party, but that isn't the end of the story: we have to actually use that information to do something about it.

Python specifically has three methods to act upon such information, the most prevalent of which is the `if` statement. The format that an `if` statement takes is the keyword `if` followed by an expression to be evaluated and a block of code to execute if (and only if) that expression evaluates to True. If the expression evaluates to False then the block of code will be completely skipped when the program is run and none of it will be executed. Put the following in a file called *second.py*:

```
x = input("Enter a number: ")
if x > '3':
    print ("You entered", x)
    print ("'x' is larger than 3")
```

Now run your program using `python second.py` and enter `2` when prompted.

```
nyx@nyx-desktop:~/umonya_notes$ python second.py
Enter a number: 2
```

Hmmmm! Why didn't our print statements execute? If, as we believe, the program executes from top to bottom, the print statements should have been executed. Obviously, that `if x > '3':` is doing something we are unaware of, so let's experiment a little more. Run the program again, but this time enter `5` at the prompt.

```
nyx@nyx-desktop:~/umonya_notes$ python second.py
Enter a number: 5
You entered 5
‘x’ is larger than 3
```

Okay! Our program just acted differently based on the input we gave it, or rather, based on the conditions determined by the input. Looking back at the last three lines of our program, there's a lot of new stuff in there that needs explanation, so we'll break it down. The if statement takes the format:

```
if <expression>:
    statement
    statement
    ...
```

What `if` does, is check whether the expression provided has a value of `True`, and if so, the statement immediately following the `if` is executed, otherwise it is not executed. Taking our previously expressed English examples:

- *If* there are between 5 and 40 people, and it's either a Friday or a Saturday, then **go to the party**
- *If* you'll get wet, **don't go outside**
- *If* it's dark in your room, and the lights are off, **turn them on**

We can now express them in Python as follows

```
if numOfPeople>5 and numOfPeople$<$40 and (day==‘Friday’
    or day==‘Saturday’):
    whatYoureDoing="going to the party"
```

```
if weather==‘Raining’ and ‘raincoat’ not in
   thingsImWearing:
    pass #In other words, don’t go outside
```

```
if dark and lightSwitch==‘off’:
    print("Go turn on your light")
```

Note the statement `pass`. Pass is another reserved word in Python, and means quite simply *do nothing*. We need it because `if` statements **require** a statement to execute if the expression they evaluate is True.

Pitfalls in Translation

We need to be aware of some common pitfalls that we will encounter when translating English language logic (forgive the oxymoron) into mathematical or computer logic. Examine the following examples:

- English: If both first name and surname are not blank
  Wrong: `if firstname and surname != ''`
  Correct: `if (firstname != '') and (surname != '')`
- English: If the premises contain more than 3 persons and no children or pets (*This is actually a very ambiguous statement*)
  Wrong: `if adults > 3 and children or pets == 0`
  Correct: `if persons > 3 and (children == 0 and pets == 0)`
- English: If the type is not 5 or 6
  Wrong: `if type != 5 or 6`
  Correct: `if type != 5 and type != 6`
  Correct: `if not (type==5 or type==6)`

Why is this the case? Well, rememeber *BODMASCNAO*? The boolean operations (`not`, `and`, `or`) have the lowest priority of all Python operations so they will happen last.

6.1 Executing Multiple Statements Conditionally

What `if` does, is check whether the expression provided has a value of `True`, and if so, the statement immediately following is executed, otherwise it is not executed.

Examining what we said previously, we notice nasty phrases like *the statement*, indicating one, singular, statement. However, most of the time we'll have a number of statements that we'll only want to execute when certain conditions have been met. Python, like most programming languages, provides us with the ability to group collections of statements into **blocks**, much like composition of expressions. Such blocks can be treated as a single statement for the purposes of `if` statements, and others we will encounter soon.

Unlike most other languages Python allows us to specify blocks using **indentation** (*spacing*). Lines of code that have the same indentation are considered to be members of the same block. Blocks can also be **nested** inside one another, such lines constituting inner blocks are also members of their encompassing outer blocks. This is best illustrated using an example. (line numbers are not part of the code)

```
 1: #this program takes three numbers as input, and
    counts how many of
 2: #those numbers is even, and how many even numbers
    are negative
 3: #if an even number is entered, if it is negative, a
     message is printed
 4:
 5: #set the counts for even and negative to 0
 6: even = 0
 7: negative = 0
 8:
 9: #get the first number
10: first = int(input("Enter the first number: "))
```

```
11: if first % 2 == 0:
12:     print("The first number is even")
13:     even = even + 1
14:     if first < 0:
15:         print("The first number is negative")
16:         negative = negative + 1
17:
18: #get the second number
19: second = int(input("Enter the second number: "))
20: if second % 2 == 0:
21:     print("The second number is even")
22:     even = even + 1
23:     if second < 0:
24:         print("The second number is negative")
25:         negative = negative + 1
26:
27: #get the third number
28: third = int(input("Enter the third number: "))
29: if third % 2 == 0:
30:     print("The third number is even")
31:     even = even + 1
32:     if third < 0:
33:         print("The third number is negative")
34:         negative = negative + 1
35:
36: print("There were", even, "even numbers and",
   negative, "of those were negative")
```

Running this program and giving the numbers 2, -4, and 7 as input produces

```
nyx@nyx-desktop:~/umonya_notes$ python illus.py
Enter the first number: 2
The first number is even
Enter the second number: -4
The second number is even
The second number is negative
Enter the third number: -7
There were 2 even numbers and 1 of those were negative
```

Firstly, let's get the indentation and nesting thing out of the way. Looking at lines 12 through to 16 we see they are more indented than the `if` statement on line 11. Thus these lines form a block of statements. Since this block is immediately after an `if` statement, all the statements in the block will only be executed if the `if` statement's expression is `True`. For the first number we gave as input 2, this means the first `if` statement evaluates to `True`, so we start executing the block. When we get to line 14 we encounter a second `if` statement. This second if statement is **nested** inside the first, because it will only be executed if the outer `if` statement (line 11) has an expression that evaluates to `True`. Lines 15 and 16 form a **nested block**, but are not executed because 2 is not negative, or more specifically, because 2 is not

less than 0. Conversely, lines 24 and 25 will be executed because -4 is less than 0. Although -7 is negative, lines 33 and 34 won't be executed, because the `if` statement on line 32 is part of the block from 30-34 that never gets executed

Despite the fact that this example is an order of magnitude larger than any previous examples, there are only two new things. The first being indentation and nesting, the second being the use of `int()` around our input functions. Remember that `input()` returns a string. So even if the user types in 2, `input()` returns not 2 but the string `'2'`. Strings and numbers don't really play nice together, especially when they're being compared to one another, so we need to convert the string into an integer. This is known as **type casting** or **coercion**. We can do it for any basic Python type ...

```
>>> float("3.1415")
3.1415
>>> str(143)
'143'
>>> bool(None)
False
>>> bool(-1)
True
>>> int(1.4142)
1
>>>
```

6.2 Nested ifs vs complex boolean expressions

Lets re-examine our party scenario. We could use nested `if`s to specify when to go to the party as such:

```
if numOfPeople>5:
        if numOfPeople<40:
                if day=='Friday':
                        print("Go to party")
                if day=='Saturday':
                        print("Go to party")
```

This method will work, but it is cumbersome. Too many nested `if`s can lead to confusion, and if you wanted to change your program later so that it printed `"You should go to the party"`, you might change it in one place but not the other. As such it would be much better to write that algorithm as follows:

```
if numOfPeople>5 and numOfPeople<40 and (day=='Friday'
   or day=='Saturday):
        print("Go to the party")
```

This is a much more elegant solution, and is preferable. You would only use nesting when you want to do slightly different things depending on the situation. When programming,

less is more. **You should almost always strive for less code** (so long as the program is understandable and still does what it should).

6.3 Specifying Execution Code for Alternate Conditions

So now we have a handle on basic `if` statements, we come up against the next hurdle. If our condition isn't met, our if block statements simply aren't executed. But what if we want something **else** executed instead of nothing when the condition isn't met. We could simply follow our first `if` with another `if` that has the opposite expression (using not), but this isn't a very elegant solution. Enter the `if ... else` statement.

```
1: first = int(input("Enter a number: "))
2: if first % 2 == 0:
3:     print(first, "is even")
4: else:
5:     print(first, "is odd")
```

Now, if `first` is not even, the program will print a message saying it is odd, instead of doing nothing. The indentation is of key importance here again. Note that `else` is at the same indentation level as the `if` and is also followed by a colon (`:`). The statement to be executed if `first` is not even, on line 5, is known as the **else clause**, as the statement on line 3 is known as the **if clause**.

Finally we have to deal with the case where things are not as clear cut, not so black and white.

```
 1: colour = input("Enter the name of a colour: ")
 2: if colour == "Red":
 3:     print("Stop!")
 4: elif colour == "Orange":
 5:     print("Try to stop if possible.")
 6: elif colour == "Green":
 7:     print("Go! Go! GO!")
 8: else:
 9:     print(colour, "Stop! You're obviously drunk.")
```

Running this a couple of times with various inputs we get

```
nyx@nyx-desktop:~/umonya_notes$ python illus.py
Enter the name of a colour: Red
Stop!
nyx@nyx-desktop:~/umonya_notes$ python illus.py
Enter the name of a colour: Green
Go! Go! GO!
nyx@nyx-desktop:~/umonya_notes$ python illus.py
Enter the name of a colour: orange
Stop! You're obviously drunk.
nyx@nyx-desktop:~/umonya_notes$
```

The first run things happen as expected; We get to line 2, the value of colour is *Red*, the expression for the `if` statement is `True`, we execute line 3 ... and then the program finishes! This is because line 4 is not part of the block constituting line 2's `if` clause, because it has a lower indentation level. Also lines 4, 6 and 8 are all still part of the same `if` statement. Let's look at the second run a little more closely to see what happens.

We get to line 2, the value of colour is *Green*. *Green* is not equal to *Red*, so we don't execute line 3. Instead we jump to the `else` ... hang on! What is `elif`? `elif` is short for *else if* and allows one to set up alternate conditions if earlier conditions haven't been met. So we get to the `elif` on line 4. *Green* is not *Orange* so we don't execute line 5, instead we go onto line 6, which is still part of the same if-else group. *Green* is equal to *Green* so that will return true and we will run line 7.

The third run illustrates nicely that in fact strings in Python are case sensitive too. Not just the language, but the data we manipulate with it is considered case sensitive. *Orange* and *orange* are different strings, hence neither the `if`, nor any of the `elif`s execute. As a last resort, the block after the `else` executes.

6.4 Formal Summary

- Conditions are specified using propositions, i.e. expressions that have a value of either `True` or `False`.
- Propositions are expressions and can be composed using other expressions combined with logical operators.
- The `if` statement is used to execute certain statements only if certain conditions are true when the `if` statement is executed.
- `if` statements can specify alternative conditions and the code to execute under those conditions using `elif`.
- `if` statements can specify what to execute if none of the conditions specified are true using `else`.
- A collection of statements can be grouped together in the same code block, by indenting them
- The complete syntax of the `if` statement follows, where sections enclosed in square braces are optional.

```
  if <expression>:
    statement
    statement
    ...
[ elif <expression>:
    statement
    statement
    ... ]
[ elif <expression>:
```

```
    statement
    statement
    ... ]
[ ... ]
[ else:
    statement
    statement
    ... ]
```

6.5 Exercises

1. Give a condition to check whether a number x is divisible by another number, y? *Hint: what does the operator % do?*
2. Given the English phrase, "If you're not on the VIP list or the staff list, you can't come in.", write down an equivalent Python/pseudo code condition.
3. Write a program that reads in a number and halves it, unless it's an odd number, in which case it must print a suitable error message.
4. Do you think the following is True or False?

   ```
   'Two'
   ```

   Evaluate it. Is the answer what you expected? How are strings compared to each other?
5. Write a program with the below code. Run it and give it 1 as an input. Does the program do as you expected? Why not? Try and fix it.

   ```
   a=1
   b=input()
   if a == b:
           print("You input 1!")
   ```

6. Write a program that asks the user to enter two numbers. If the second number is not zero, print the quotient (the one number divided by the other) of the two numbers, otherwise print a message to the effect of not being able to divide by zero.
7. What does the following equate to? Try figure it out for yourself before you code it up. If you're still not sure why the answer is as it is, try breaking it up into smaller bits and evaluate them individually.

   ```
   True and not True or (True or False and False) and
      False == (not True or False)
   ```

8. Write a program that asks the user to enter 4 numbers, and prints the smallest number entered.
9. Write a program that asks the user to enter 3 numbers, and prints the largest even number entered, and displays a suitable error message if no even numbers are entered, or if there is no one largest number (i.e. if the two biggest numbers are the same)

10. Write a program that reads in a sentence and a keyword and tells you if that keyword is in that sentence.

11. Write a program that asks the user to enter a four digit year, and prints out whether that year is a leap year or not.
Hint: It's a leap year if the year is divisible by four, except if it's divisible by 100, in which case it's not, except if it's divisible by 400, in which case it is. ie. 1900 was not a leap year but 2000 was.

12. Write a program that asks for the names, and the birthdays of two people. (Ask for year, month and then date of birth) The program must determine which of the two people is oldest.

13. Write a program that asks the user to enter 3 names, then outputs them sorted alphabetically.

```
import random
a=random.randrange(10)
print (a)
```

14. The above code prints a random number between 0 and 10. Use the above code in a program that asks the user to guess what number the computer is "thinking of". Then tell the user if he was right, or wrong, and if so, whether his guess was too high or too low.

15. Write a program that asks the user to enter a number, then prints out all the numbers from 1 to 10 by which the entered number is divisible.

16. Think about that last program you wrote and how much of the code was repeated. Write a better version of the same program in your own pseudo-code.

Chapter 7

While loops

7.1 The while statement

The last program we wrote, to count the number of even and negative numbers entered by the user, was already getting pretty large if one measures program size in lines of code. And it only handled three inputs! What if we wanted a hundred inputs. Cutting and pasting would work, but making those small changes to each section of code dealing with a specific number would be tedious at best, and error prone at worst. Fortunately, computers excel at repetitive tasks. Enter the `while` statement:

> The `while` statement executes a statement, or block of statements, repeatedly, as long as a given expression is `True`.

So let's rethink our previous problem. Previously, we would have described the problem as:

> The user enters three integers. The program outputs how many of those integers were even, and how many were both even and negative. Also, if the number entered is even, a message stating the number entered is even is printed. Similarly a message is printed if the number is both even and negative, indicating the number is negative.

But we are not happy with just three numbers, so let's re-describe the behaviour we want from our program

> The user may **enter** a number. **Until** the user enters a blank line, the program continues accepting numbers. After a blank line has been entered the program **outputs** the number of numbers entered by the user **which** were even, as well as the number of numbers **which** were **both** even **and** negative.

Much like how in junior school we were taught to look for key words in word problems to help us formulate the problem mathematically, we can and should do the same with descriptions of problems and their translation into program code. Everything in programming comes down to one of three *structures*...

- A sequence of steps

- A condition that selects which sequence of steps to execute
- A repetition of a sequence of steps

In the problem description above, we can identify some key words already hint at the structures we should use.

enter Relates to input – `input()`

until Indicates **repetition** based on the fulfilment of a **condition** – `while`

output Indicates **output** – `print()`

which Indicates a **condition without repetition** – `if`

both ... and ... Indicates a **composition of conditions** – `and`

We'll now step through the problem description, sentence by sentence and convert it into a Python program, using the keywords as hints, and the tools we have already come across. Also we'll introduce the syntax of the while statement.

> The user may enter a number.

Looking at the keywords and their hints, we want to use the `input` function here. This is true of any input from the keyboard generally. Thus, our first line of code will be

```
number = input("Enter a number: ")
```

> Until the user enters a blank line, the program continues accepting numbers.

Okay, from the hints we see we should be using a `while` statement, but **beware**! We don't have an 'until' statement in Python, and in English until and while are opposites, so we just need to transform the problem slightly so that we now have

> **While** the user enters a line **that is not blank**, the program continues accepting numbers.

This translates directly into Python code which we append to what we've already got to produce

```
number = input("Enter a number: ")
while number != '':
    number = input("Enter another number (or nothing to
       finish): ")
```

Note the prompt has changed to tell the user what to do to exit the program, namely enter a blank line rather than a number. Also note the format, or syntax, of the `while` statement.

```
while <expression>:
    statement
    statement
    ...
```

See how we *indent* statements we want executed repetitively and conditionally under the `while` statement, in the same way as we do with `if` statements. The `while` statement checks the expression, and if it is `True`, will execute those indented statements once, after which it will check the expression again, and execute the statements again, etc ... If the expression is ever `False`, execution of the program continues at the first un-indented statement after the indented block.

There's a little more to the while loop than pure syntax. Every loop needs three things; A start point, an end point, and a way to get from one point to another. More importantly there needs to be a relationship between these three things, in the form of a variable. The start point takes the form of assigning a value (the starting value) to a variable, which we'll call the counter. The while loop's condition specifies the stop point of the loop, by specifying a condition under which the loop should terminate **in terms of the counter variable**. Finally, the loop needs a way to get from start to finish in a stepwise manner, i.e. a way to take a single step. This means the value of the counter variable must change **inside** the repeated block of statements, otherwise value of the loop condition won't change, and the loop will repeat forever. Commonly, the statement that changes the counter's value is placed at the *end* of the repeated block of statements, because this means it is changed immediately before the counter variable is checked in the loop condition again. This gives us the pattern:

```
counter = <start_value> #Initialise the counter
while <expression>: #<expression> specifies when the
   loop will stop in terms of counter
    statement
    statement
    ...
    counter = <new value of counter> #changes the value
       of counter
```

After a blank line has been entered the program outputs the total amount of numbers entered by the user which were even.

Now we haven't kept a record of the numbers entered so how can we tell how many were even? The solution is to keep a count as even numbers are entered. How do we store a value? **Variables**! We need to know **how many** numbers were even, indicating quantity, indicating a number, i.e. an integer. So let's create a new integer to use whilst counting, and call it `even`. But to create a new variable we have to give it a value! What value can we give even, if we don't know how many even numbers the user will enter? Well we do know how many even numbers there are before the user has entered any numbers; there are 0 even numbers. So let's put an assignment statement to that effect into our program in the right place (before a number is entered).

```
even = 0
number = input("Enter a number: ")
while number != '':
    number = input("Enter another number (or nothing to
       finish): ")
```

Now at the end of our program we still have 0 even numbers, because we haven't changed the value of `even`. We wish to count how many numbers that are entered are even, which means we need to increment (increase by one) every time an even number is entered. Firstly, how do we distinguish even numbers from odd? We use an `if` statement, because this is a condition. Secondly, where to we actually place the counting statements? Here's a suggestion:

```
even = 0
number = input("Enter a number: ")
while number != '':
    if int(number) % 2 == 0: #if number is even
        even = even + 1
    number = input("Enter another number (or nothing to
       finish): ")
```

A few things about our two new lines deserve mention. Firstly, as we are already familiar with, `input()` returns a string, so we need to **type cast** to an integer before we can test whether it is even. Then there's the condition itself. The definition of an even number is a number divisible exactly by 2, i.e. without remainder. The '`%`' operator returns the remainder of a division, and is thus perfectly suited to the job of testing whether a number is even or odd. If the remainder of the division is 0, the number is even. Also, we have nested the `if` statement within the `while` statement, so it may be executed multiple times, once each time a number is entered, to test the number. We have put it before the input function within the `while` statement because we don't want to test an empty string (blank line) for evenness in the case where the user has not entered a number and wishes to finish up. This way the `while` statement's expression tests whether the input is a blank line, before we convert to an integer and test for evenness.

as well as the number of numbers which were both even and negative.

The last piece of the problem description says we should also count how many of the input numbers are negative as well as even. And we need to put some output at the end once we've counted everything. Well, this is very similar to the previous segment, so let's recycle the idea and see what we get...

```
even = 0
negative = 0
number = input("Enter a number: ")
while number != '':
    if int(number) % 2 == 0: #if number is even
        even = even + 1
        if int(number) $<$ 0:
            negative = negative + 1
    number = input("Enter another number (or nothing to
       finish): ")
print("There were", even, "even numbers, of which",
   negative, "were also negative")
```

So we've included a new variable, `negative`, which counts the number of negative entries, but only if those entries are even. Why is this? We haven't specified (`int(number) % 2`

`== 0 and int(number) < 0`). Instead we have nested the test for negativity inside the test for evenness. This means a number will only be tested, and thus potentially counted, for negativity if it has already been found to be even.

7.2 The break statement

Looking at the `while` statement, it seems that once we're in a block of statements to be executed repeatedly, known as a **loop**, we can't get out of the loop except when the expression after `while` (known as the **loop condition**) is `False`. Python provides us with a statement for breaking out of a loop, conveniently called `break`. Suppose we wanted our program for counting even numbers to end not only when a blank line was entered, but also if the user enters the string `'quit'`. We could simply add two lines ...

```
even = 0
negative = 0
number = input("Enter a number (or nothing or 'quit' to
   quit): ")
while number != '':
    if number == 'quit':
        break
    if int(number) % 2 == 0: #if number is even
        even = even + 1
        if int(number) $<$ 0:
            negative = negative + 1
    number = input("Enter another number (or nothing to
       finish): ")
print("There were", even, "even numbers, of which",
   negative, "were also negative")
```

Again we check what the user has entered, and if it is the string `'quit'`, we `break` out of the loop, meaning execution continues at the `print()` function.

7.3 The continue statement

If we wanted to get picky and consider 0 to not be even, we would have to modify our code so that it doesn't count a '0' entry as even or negative. We can't simply break out of the loop, because the user may want to enter more numbers after the '0'. We could enclose the entire test for evenness in an `if` statement that makes sure the number entered is not 0, but Python provides us with a more elegant solution, the `continue` statement. The `continue` statement jumps the flow of execution immediately back to the loop condition, at which point normal loop execution flow resumes.

```
even = 0
negative = 0
```

```
number = input("Enter a number (or nothing or 'quit' to
   quit): ")
while number != '':
    if number == 'quit':
        break
    if number == '0':
        number = input("Enter another number (or nothing
            or ''quit' to quit): ")
        continue
    if int(number) % 2 == 0: #if number is even
        even = even + 1
        if int(number) $<$ 0:
            negative = negative + 1
    number = input("Enter another number (or nothing or
       'quit' to quit): ")
print("There were", even, "even numbers, of which",
   negative, "were also negative")
```

7.4 else clauses in while loops

`while` loop statements may also have an else clause, which is executed when the loop terminates when the condition becomes `False`, but not when the loop is terminated by a `break` statement.

```
while <expression>:
    <statement>
    [statement]
else:
    <statement>
    [statement]
```

7.5 Exercises

Given the code ...

```
i = 1
while i
```

1. How many lines of output will the above code produce?
2. What needs to be done to correct the program?
3. Write a program that outputs the word 'repeat' 100 times, each on a line of its own.
4. Write a program that prints the numbers from 1 to 10 on the screen, on a single line, ending with a new line.

5. Write a program that asks the user for a number and then prints the numbers from 1 to the number they entered.
6. Write a program that asks the user for a number and then prints the sum of numbers from 1 to the number they entered.
7. Write a program that asks the user for two numbers and then prints the sum of numbers from the lowest number entered to the highest number entered.
8. Write a program that asks the user to enter a sequence of numbers, ending with a blank line. Print out the smallest of those numbers.
9. Write a program that asks the user to enter a sequence of numbers, ending with a blank line. Print out the average of those numbers.
10. Write a program that prints the numbers from 1 to 100, 10 per each line.

Chapter 8

Lists

8.1 The Necessity of lists

Let us for the sake of originality work on a new problem.

> A lecturer for an introductory programming course wants to record some information about his students. He wants to keep a record of each student's name, their primary field of study, and a brief description of their current research project. He would like to be able to print this out in a nicely formatted way. His class has 19 students.

For each student we need to record their name, field of study, and a description of their research project. This means three values we need to store for each student, or 38 variables in total. Surely there's a better way. There is, and Python yet again provides us with the perfect tool for job, the **list**!

A list is a **type** of variable. To create a new variable of type list in Python simply assign a variable to a list. Lists are formed using square brackets surrounding a comma separated sequence of the elements of the list. Example:

```
>>> mylist = [1, 2, 3]
>>> anotherlist = ["Alice", "Bob", "Carl", "Mallory"]
>>> mixed = ["a string", 143, [341]]
>>> empty = []
>>>
```

- Lists contain elements, e.g. `mylist` contains three elements, namely 1, 2, and 3.
- Lists have length, being the number of elements they contain, e.g. `mylist` has length 3, `anotherlist` has length 4
- We can determine the length of a list using Python's built in `len` function

```
>>> len([1, 2, 3])
3
```

- The elements of a list can be expressions of any type (including another list) and different elements in the same list can be of different types, e.g. `mixed` has three elements; the first element is a string (`"a string"`), the second element is an integer (`143`), and the third element is another list (`[341]`).
- Lists can contain no elements, in which case they are known as *empty*, e.g. `empty` contains no elements, but is still a list!
- Individual elements of a list are indexed according to their position in the list. The first position has an index of 0, the second position an index of 1, etc...
- We can access a specific element in a list by using a special operator in an expression with the format `list[index]`. This expression evaluates to the element in the index'th position of the list. e.g. `mylist[0]` is 1, `mylist[1]` is 2, `mixed[2]` is `[341]`.

Now that we know about lists, we could use a list to represent the students names, another to represent their fields of study, and a third to represent their project descriptions. So we might have

```
#a small program to record info about students

name = ["Ayanda", "Ben", "Carl", "Dumisani"]
field = ["Astronomy", "Biochemistry", "Cancer Research",
    "Maths"]
description = [
    "The search for black holes",
    "Engineering a better yeast for brewing beer",
    "Better pain management in palliative care",
    "Finding a polynomial time solution for NP-complete
       problems"
]
```

8.2 Lists in Detail

We've already learnt a little about about lists, but there are many more features to lists, and sadly a few intricacies we need to know about. Firstly let's deal more explicitly with the operations that can be performed on lists. If both `a` and `b` are lists, `v` is any expression, and `i`, `j`, and `k` are integers then

- `a + b` concatenates (joins) two lists.
  It returns a new list that contains all the elements of list `a` and then list `b`.

```
>>> a = [1,2,3]
>>> a+[4,5]
[1, 2, 3, 4, 5]
>>>
>>> [4,5]+a
[4, 5, 1, 2, 3]
```

- `a * i` repeats a list.
  It returns a new list with that has all the elements of `a` but repeated `i` times.

```
>>> a*2
[1, 2, 3, 1, 2, 3]
>>>
>>> [1]*2+2*[2]
[1, 1, 2, 2]
>>>
>>> [[1]*2,[2]*2]
[[1, 1], [2, 2]]
>>>
>>> [1]*2*3
[1, 1, 1, 1, 1, 1]
>>>
```

- `a[i]` retrieves the `i`'th element of `a`.

```
>>> a[0]
1
```

- `a[i:j:k]` slices the list.
  It returns a copy of the list `a` from its `i`'th element to its `(j-1)`'th element, taking only every `k`'th element. If `k` is omitted (left out), every element in the range is taken. If `i` or `j` are omitted they default to beginning of list and end of list respectively.

```
>>> a[1:3]
[2, 3]
>>> a += [4, 5]
>>> a[0:6:2]
[1, 3, 5]
```

- `a[:i]` yields a slice from the beginning of `a` to the `(i-1)`'th element of `a`.

```
>>> a[:3]
[1, 2, 3]
```

- `a[i:]` yields a slice from the `i`'th element to the end of the list.

```
>>> a[3:]
[4, 5]
```

- `a[:]` yields a slice of the entire list. More importantly it yields a **copy** of the list!

```
>>> a[:]
[1, 2, 3, 4, 5]
>>>
```

Note that theses operators each return newly created lists, and do not modify the *operand* lists (that is, the list that this operations are performed on) in any way.

8.3 Assignment to elements of a list

Lists can have their elements changed. This sounds a little weird, but consider it this way. The value of the integer 1 cannot change. We can change the value indicated by a variable `i` (which might be 1) to something else, but this doesn't change the value of the integer 1. Lists are different, because the value reflects a container of multiple other values, and we can remove, add, or replace those inner values, i.e. list elements. This changes the value of the list itself. Practically, this means we can assign values to specific elements of a list.

```
>>> a = [1, 2, 3]
>>> a[0]
1
>>> a[0] = 0
>>> a
[0, 2, 3]
>>>
```

Not only can we assign individual elements of a list a new value, but we can assign whole slices new values. The funky thing is, we can replace a specified slice of a list with a list of any length, effectively extending or shrinking the list in the process.

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

8.4 Comparing Lists

The comparison of lists to other lists, and other types is also slightly different to the comparison of simple types (int, float, bool). Specifically:

- `a == b` checks two lists are the same.
  It returns true if all the elements of `a` are the same as the elements of `b`.
- `a < b` yields Less Than.
  Note that lists are compared in sequence order, meaning that `a` is less than `b` if `a`'s first element is less than `b`'s first element. If `a`'s first element and `b`'s first element are equal, the second elements are checked, etc...
- `v in a` is True only if the list a contains an element whose value is that of the expression `v`.

- `a is b` is True only if `a` and `b` are the same list.

Note the final point, namely the `is` operator. This is one of those intricacies we mentioned earlier. The contents of a list can be changed. This is not true of an integer for example. One cannot change the contents of the integer 1. This introduces a slight complexity. If `a` is 1 and `b` is 1 then `a` and `b` are the same. But if `a = [1]`, and `b = [1]`, they **are not** the same. They may be equal, but are not the same list. They are different lists. Again, examples to the rescue

```
>>> a = [1,2]
>>> b = [1,2]
>>> a == b
True
>>> a is b
False
>>>
>>> b[0] = 3
>>> b
[3, 2]
>>> a
[1, 2]
>>>
```

```
>>> a = [1,2]
>>> b = a
>>> a == b
True
>>> a is b
True
>>> b[0] = 3
>>> b
[3, 2]
>>> a
[3, 2]
>>>
```

Note how the second line of the second example contradicts our basic idea of assignment, which was: The assignment statement assigns the value of the expression on the right to the variable on the left. But clearly, we are not just assigning the value (`[1,2]`), because later when we change the first element of `b` to 3, we also appear to change the first element of `a`. Weird. There is a technical explanation for this, but is really not worth knowing at this point as it is more likely to confuse than clarify. If you are interested, ask one of thew tutors to explain this. What this does mean however is that we need to be aware of the special case of assignment to a single variable of types that can be changed! When, in an assignment statement `a = b`, if `b` is of **mutable type**(a type that can have its values changed), the assignment statement **makes `a` a synonym of `b`**, i.e. they become different names for the same variable.

Also note how we assigned a value to an element of the list, changing some of it's contents. Python is quite advanced in terms of how it can handle assignments involving lists, so let's

explore its features a bit. If `a` is the list `[1, 2, 3]`, `b` is some expression of any value, and `l` is some expression of type list.

- `a[0] = b` assigns the value of `b` to the 0'th element of `a`. (The mutable type special case applies)
- `a[0:2] = l` removes the slice range specified on the left and replaces the removed elements with the contents of `l`. This does **not** make `a` and `l` synonyms.

If we wish to make a copy of a list that is not a synonym, we assign to the slice of the full list, e.g. `a = b[:]` (where `b` is a list)

8.5 Lists as objects

In addition to the operators that can act on lists, list are objects. An **object** is a term used in Computer Science to refer specifically to something which has both code and data associated with it directly. For now, a complete understanding of objects is not required, but in the meanwhile we need to be aware of some syntactical features they provide.

Objects have code associated with them in functions called **methods**. Methods are bound to every variable of an object type individually, meaning if two different variables, `a` and `b`, are of the same type which is an object type, then `a` and `b` both have the same set of methods but their methods are distinct from one another. When `a` calls a method, that method acts only on `a`, and when `b` calls that same method, the method will act only on `b`. Read it again, it makes sense. In fact examples will prove it's quite intuitive.

```
>>> a = [1, 2, 3]
>>> b = ["A", "B", "C"]
>>> a.index(2)
1
>>> b.index(2)
Traceback (most recent call last):
  File "", line 1, in
ValueError: 2 is not in list
>>> b.index("C")
2
```

Looking at the examples we notice there is some new notation, of the form <`list`>.<`method`>(). This *dot notation* indicates that we are obtaining a method of the list object (called index in the case of the example), and our use of brackets indicates we wish to call it. Methods are simply functions associated with a particular type.

Note that `a.index(2)` and `b.index(2)` return different results. In fact the `b.index` case produces an error. Obviously they are doing different things, despite being given the same parameter (2). What the index method does is search for a given value in the elements of **its** list. When called using `a.index` its list is `a`, when called using `b.index` its list is `b`. Since the value 2 is an element of the list `a`, index returns its index position (1), meaning it can be found in the second position in the list. But the value 2 is not an element of the list `b`, hence index will cause an error.

That being said, let's round off our knowledge of lists by going through the list methods that will be useful to us.

- <list>.append(<expression>)
  Appends the value of <expression> to the end of <list>.

```
>>> a = []
>>> a.append('one')
>>> a
['one']
>>>
```

- <list>.count(<expression>)
  Returns the number of elements in <list> with same value as that of <expression>.

```
>>> a = ['a', 'c', 'g', 't', 't', 'a']
>>> a.count('a')
2
>>>
```

- <list>.index(<expression>)
  Returns the index of the first element in <list> that has the same value as <expression>.

```
>>> a = ['a', 'c', 'g', 't', 't', 'a']
>>> a.index('t')
3
>>>
```

- <list>.insert(<index expression>, <object expression>)
  Inserts the value of <object expression> **before** the element at the position indicated by <index expression>.

```
>>> a = ['a', 'c', 'g', 't', 't', 'a']
>>> a.insert(2,'newitem')
>>> a
['a', 'c', 'newitem', 'g', 't', 't', 'a']
>>>
```

- <list>.pop()
  Removes the last element from the end of the list, and returns it's value.

```
>>> a = ['a', 'c', 'g', 't', 't', 'a']
>>> a.pop()
'a'
>>> a
['a', 'c', 'g', 't', 't']
>>>
```

- <list>.remove(<expression>)
  Removes the first element in <list> that has the same value as <expression>.

```
>>> a = [‘a’, ‘c’, ‘newitem’, ‘g’, ‘t’, ‘t’, ‘a’]
>>> a.remove(’newitem’)
>>> a
[‘a’, ‘c’, ‘g’, ‘t’, ‘t’, ‘a’]
>>>
```

- <list>.reverse()
  Reverses the order of elements in <list>.

```
>>> a = [‘alpha’, ‘beta’, ‘gamma’, ‘delta’]
>>> a.reverse()
>>> a
[‘delta’, ‘gamma’, ‘beta’, ‘alpha’]
>>>
```

- <list>.sort()
  Sorts the elements of <list>.

```
>>> a = [‘delta’, ‘gamma’, ‘beta’, ‘alpha’]
>>> a.sort()
>>> a
[‘alpha’, ‘beta’, ‘delta’, ‘gamma’]
>>>
```

8.6 Exercises

1. Given the list `l = [‘There’, ‘are’, 9000000, ‘bicycles’, ‘in’, ‘Beijing’]`
   (a) How many elements does the list have? How would one find this out programatically?
   (b) What is the index of the non-string element?
   (c) What is the output of `print(l[1:4])`?
2. What is the value of `[1, 3] + [2, 4]`?
3. What is the value of `[1, 3] + [[2, 4]]`?
4. What is the difference between appending a value to a list and adding the same value within its own list using the concatenation operator?
5. What is the slice notation to extract the word `‘mickles’` from the string `‘How many mickles are there in a muckle?’`
6. Write a program that outputs the position of every `‘t’` in the list `[‘a’, ‘c’, ‘g’, ‘t’, ‘t’, ‘a’, ‘t’]`. Use the index method, don’t do it manually.
7. Write a program that asks the user to enter a sequence of up to 5 x:y coordinates with both x and y in the range 0 to 4, ending their sequence entry by providing a blank line for the x coordinate. Then display a five by five grid of ‘#’ characters, with the points

in the grid entered by the user left blank. Assume x increases from left to right, and y increases from top to bottom. Example input/output is given ...

```
Coordinates range from 0 to 4!
Please enter pair of coordinates (x:y), leave x
   blank to terminate sequence.
X> 3
Y> 3
Please enter pair of coordinates (x:y), leave x
   blank to terminate sequence.
X> 4
Y> 1
Please enter pair of coordinates (x:y), leave x
   blank to terminate sequence.
X> 1
Y> 4
Please enter pair of coordinates (x:y), leave x
   blank to terminate sequence.
X>
#####
####
#####
### #
# ###
```

Chapter 9

Tuples

9.1 Tuples as immutable lists

In many ways tuples and lists are very similar. In fact tuples are basically immutable lists, that is, lists which cannot be changed. All the same operators work on them: concatenation, repetition, subscription, and slicing. All the comparison operators work in the same way: equivalence, less than, greater than, `is`. However, **assignments to tuples cannot be made** as they are immutable, or unchangeable. Once a tuple has been formed, it cannot be changed. New tuples can be formed from others using concatenation, repetition, etc... but the operand tuples remain unchanged by the operators.

9.2 Forming a tuple

Tuples are technically formed in Python using a comma, although most often they are found recorded in round braces. Tuples can be of any length, and have their elements indexed from `0` to `len(tuple)-1`.

```
>>> 1, 2
(1, 2)
>>> 1,
(1,)
>>> (1)
1
>>> (1, 2)
(1, 2)
>>> ("Tuples", "can", "have", "elements", "of", "more",
   "than", 1, "type")
('Tuples', 'can', 'have', 'elements', 'of', 'more', '
   than', 1, 'type')
>>>
```

9.3 Using tuples

Breaking it down we get to the old grind of operators, and in the case of tuples a few new ideas. Let's get the operators out of the way! If both `a` and `b` are tuples, `v` is any expression, and `i`, `j`, and `k` are integers then

- `a + b` concatenates two tuples.

```
>>> a = (1,2,3)
>>> a+(4,5)
(1, 2, 3, 4, 5)
>>>
>>> (4,5)+a
(4, 5, 1, 2, 3)
```

- `a * i` repeats a tuple.

```
>>> a*2
(1, 2, 3, 1, 2, 3)
>>>
>>> (1,)*2+2*(2,)
(1, 1, 2, 2)
>>>
>>> ((1,)*2,(2,)*2)
((1, 1), (2, 2))
>>>
>>> (1,)*2*3
(1, 1, 1, 1, 1, 1)
>>>
```

- `a[i]` retrieves the `i`'th element of `a`.

```
>>> a[0]
1
```

- `a[i:j:k]` slices a tuples, or returns a copy of the tuple `a` from its `i`'th element to its `(j-1)`'th element, taking only every `k`'th element. If `k` is omitted, every element in the range is taken.

```
>>> a[1:3]
(2, 3)
>>> a = (1, 2, 3, 4, 5)
>>> a[0:6:2]
(1, 3, 5)
```

- `a[:i]` yields a slice from the beginning of the tuple to `(i-1)`'th element of `a`.

```
>>> a[:3]
(1, 2, 3)
```

- `a[i:]` yields a slice from the `i`'th element to the end of the tuple.

```
>>> a[3:]
(4, 5)
```

- `a[:]` yields a copy of the entire tuple.

```
>>> a[:]
(1, 2, 3, 4, 5)
```

The comparison of tuples to other tuples happens in the same way as lists to lists.

- `a == b` yields Equivalence
- `a < b` yields Less Than. Note that tuples are compared in sequence order, meaning that `a` is less then `b` if `a`'s first element is less than `b`'s first element. If `a`'s first element and `b`'s first element are equal, the second elements are checked, etc...
- `v in a` is True only if the tuple `a` contains an element whose value is that of the expression `v`.
- `a is b` is True only if `a` and `b` are the same tuple.

If tuples have a subset of the functionality of lists, why ever use them? What can they do that lists cannot. Well, actually they provide a very convenient way to package multiple variables together for passing to a function as a parameter, or more likely returning from a function. Technically speaking functions can only return one value, but since a tuple is a single 'value' no matter how many elements it contains, a tuple could be used in a function that returns a two-component piece of data, for example an x:y coordinate. Even better, is that Python *implicitly unpacks tuples*. By unpack we mean the assignment of multiple variables at once to their respective (based on position) elements in a tuple, for example

```
>>> t = (1, 2, 3)
>>> a, b, c = t
>>> a
1
>>> b
2
>>> c
3
>>>
```

And since tuples are formed by a comma, this provides us with a nifty little method to swap the values of two variables,

```
>>> a = 1
>>> b = 2
>>> b, a = a, b
>>> a
2
>>> b
1
```

9.4 Exercises

1. What is the difference between concatenation and addition of tuples? Why is the Python plus operator for tuples a concatenation operator instead of an addition operator?
2. The code `a, b = b, a` swaps the values of `a` and `b`. What would happen if added a third variable to the mix, `c`, as in `a, b, c = b, c, a`? What would happen if one repeated the statement indefinitely?
3. Given a variable with the value of the empty tuple (`a = ()`), how does one 'append' an element to `a`?
4. Write a program that accepts a list of numbers terminated by a blank line. Print out the entered numbers as elements of a tuple, in the order they were entered.
5. Given a list of tuples each specifying a subject name and a grade symbol ('A' - 'F') in the form `[('Maths', 'D'), ('Comp Sci', 'B'), ('English', 'C'), ('Xhosa', 'A'), ('Science', 'B'), ('History', 'E')]`:
   (a) Write a program that prints out the subject with the highest mark.
   (b) Write a program that outputs each subject and the grade symbol in the format 'subject: symbol', with each subject on a single line.
   (c) Write a program that prints out a tab separated list of subjects on the first line, and the corresponding grades, also tab separated on the second line.
6. Write a program that accepts a list of numbers terminated by a blank line, and stores the result in a tuple, `a`. Repeat the process to form a second user inputted tuple, `b`, making sure there are the same number of elements in `b` as in `a`. Print out the result of the *mathematical addition* (not concatenation) of the two tuples as a tuple.
7. Write a program that reads a name and an age for a person, until the name is blank. As each name age pair is entered, store names in a list, and ages in another. Print a list of tuples of paired names and ages.
8. Write a program that reads a name and an age for a person, until the name is blank. Once all names have been entered, present the user with an option to list the entered people in alphabetical order, or in descending age order. For either choice, list each person's name followed by their age on a single line. Make sure you output the correct age for the correct person.

Chapter 10

For loops

10.1 Loops over Lists

In lesson 8 we introduced the concept of lists.

A list is a type of variable that contains other elements. For instance we could create a shopping list.

```
>>> shopping=[‘Milk’,‘Bread’,‘Eggs’,‘Motherboard’,‘
   Cheese’,‘Jacket’]
```

In order to view the individual contents of that list, we could refer to them by index

```
>>> shopping[0]
‘Milk’
>>> shopping[3]
‘Motherboard’
```

In order to view all the items of our list, we could manually type in all the indexes, but what happens if you add more items later, or if you have 100 items? A better idea would be to use a `while` loop to change the index. We set the index to 0 to begin with, then while the index is less than length of the list, we print out the item in the list referenced by the index and increment the index.

```
>>> index=0
>>> while (index < len(shopping)):
...     print(shopping[index])
...     index+=1
...
Milk
Bread
Eggs
Motherboard
Cheese
Jacket
```

That works, but, once again Python provides us a better tool for the job in the form of the `for` statement.

The `for` statement is also a looping statement, like the `while` statement, except that it loops over the elements of a list rather than until a boolean expression is false. It looks like this

```
for <element> in <list>:
    statement
    statement
    ...
```

Where *element* is a variable name (not necessarily 'element'), which may be as yet unassigned (the `for` statement will assign a value to it), or may be an already existing variable, in which case a new value will be assigned to that variable by the for statement. `list` is an expression whose value is of type list.

What this does exactly is execute the indented statements **once for each element in the list** `list`. Each time the statements are executed, the variable in the position of `element` is assigned the value of the next element in `list`, starting with the first.

So, for example, the following program

```
#a small program to display your shopping list
shopping=['Milk','Bread','Eggs','Motherboard','Cheese','
   Jacket']
for i in shopping:
    print("Item:",i)
```

produces the following output

```
Item: Milk
Item: Bread
Item: Eggs
Item: Motherboard
Item: Cheese
Item: Jacket
```

Well, that's pretty cool. Notice how the variable `i` changes on every iteration (every repeat of the loop). But what if we wanted to number the list too, or have two lists, one with the item and one with the price. What we could do is have a list of indexes, and use the for loop to go through them, printing out both the index and the item in the shopping list that corresponds to that number. We could do that as follows:

```
#a small program to display a numbered shopping list
indexes=[0,1,2,3,4,5]
shopping=['Milk','Bread','Eggs','Motherboard','Cheese','
   Jacket']
prices=['8','6','10','800','30','150']
for i in indexes:
    print("Item number",i,"is",shopping[i],"and it costs
        R",prices[i])
```

produces the following output

```
Item number 0 is Milk and it costs R 8
Item number 1 is Bread and it costs R 6
Item number 2 is Eggs and it costs R 10
Item number 3 is Motherboard and it costs R 800
Item number 4 is Cheese and it costs R 30
Item number 5 is Jacket and it costs R 150
```

Note that if you were printing this for your mother, she might not understand why milk is the 0th item and not the 1st, so you might want to `print(i+1)` *so that it seems to start at 1.*

That works alright. But if we were dealing with larger lists, like of 19 items, or even thousands of items, we really don't want to have to type out the list of possible indexes ([0, 1, 2, ..., 19]) every time. Python again comes to the rescue with another built in function, `range`. `range` returns a list of numbers within a given range, and defined as follows

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

What it does is returns a list of integers, starting with the number given by `start` (or 0 if `start` is not given), up to but not including the number given by `stop`. If `step` is given, the list will only provide every `step`'th integer in the sequence. Examples probably illustrate this better ...

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(4,10)
[4, 5, 6, 7, 8, 9]
>>> range(-3,3)
[-3, -2, -1, 0, 1, 2]
>>> range(0,10,2)
[0, 2, 4, 6, 8]
>>> range(10,20,3)
[10, 13, 16, 19]
>>>
```

So we can change our previous program slightly, as follows

```
#a small program to display a numbered shopping list
shopping=['Milk','Bread','Eggs','Motherboard','Cheese','
   Jacket']
for i in range(len(shopping)):
    print("Item number",i,"is",shopping[i])
```

The `for` loop isn't only used with lists. The `for` loop can be used with any iterable structure, i.e. any structure which can be indexed. One such structure is the string, which you'll examine in depth in the next chapter. As has been mentioned previously, the string is basically a list of characters, and hence you can use a `for` loop to iterate through those characters. So for example to print every letter of a word on a new line:

```
>>> for i in "Any Word":
...     print(i)
...
A
n
y

W
o
r
d
```

The `for` loop also works on parts sections of lists, and edited lists, here are some examples:

```
>>> for i in ["Ford","Mercedes","BMW","Audi","Toyota","
   Nissan"][2:5]:
...     print(i)
...
BMW
Audi
Toyota
>>> for i in "Experts Exchange"[6:10]:
...     print(i,)
...
s   E x
>>> for i in ["Animals",3,"blah"]+[2,True,"for you"]:
...     print(i, " | ",)
...
Animals  |  3  |  blah  |  2  |  True  |  for you  |
>>> for i in "Hello how are you doing?"[::2]:
...     print(i,)
...
H l o h w a e y u d i g
```

10.2 else clauses in for loops

`for` loop statements may also have an `else` clause, which is executed when the list is finished, but not when the loop is terminated by a `break` statement.

```
for <variable> in <list>:
    <statement>
    [statement]
else:
    <statement>
    [statement]
```

10.3 Exercises

1. Write a program that asks the user to enter two (whole) numbers and outputs the product. However, do so without using * but rather by repeatedly adding.
2. Write a program that prints out the string `"repeat"` 100 times.
3. Write a program that asks the user for their name, and a number, then prints out the user's name, with some motivational comment, that many times.
4. What is an expression, using the `range` function, that yields a list of 10 consecutive integers starting at 0?
5. What is an expression, using the `range` function, that yields a list of 100 consecutive integers starting from 1?
6. What is an expression, using the `range` function, that yields a list of 100 consecutive *even* integers starting from 2?
7. Write a program that prints the *odd* numbers from 1 to 100 each on its own line.
8. The triangular numbers are the numbers 1, 3, 6, 10, 15... The n'th triangular number is the sum of all the integers from 1 till n. e.g. the 4th triangular number is 1+2+3+4=10. Write a program that accepts an integer value for n from the user, and prints the n'th triangular number.
9. Write a program that accepts words from the user, storing them in a list, until the user enters the word "end". The program must then print out all the words in reverse order.
10. Modify your answer to question 10 from "Flow Control: Conditionals", i.e. print out the numbers from 1 to 10 by which a user entered number is divisible, to use a `for` loop instead of multiple `if` statements.
11. Write a program that asks the user for the height of a triangle. It must then print out a right handed triangle, with the right-angle on the bottom left, made of asterisks (*) of a height equal to the number entered. Example input/output follows ...

```
Enter triangle height: 3
*
**
***
```

```
Enter triangle height: 5
*
**
***
****
*****
```

12. Do the same as above, except now with the right-angle in the top right, and after you have printed the first triangle, you must start all over again until the user enters a blank line for input. Example ...

```
Enter triangle height: 3
***
 **
  *
Enter triangle height: 5
*****
 ****
  ***
   **
    *
Enter triangle height:
```

13. Write a program that prints out a 'Christmas' tree shape from asterisks. The program should ask the user for the height of the tree, in lines, and the tree should have a stalk of two lines regardless. If the user enters a height, the tree should be printed out, and the user should be prompted for another height until they enter a blank line.

```
Enter tree height: 4
   *
  ***
 *****
*******
   *
   *
Enter tree height: 6
     *
    ***
   *****
  *******
 *********
***********
     *
     *
```

Chapter 11

Strings

11.1 Strings as Sequences

Strings can be thought of as sequences (as lists are sequences) of characters. As such, many of the methods that work on lists work on strings. Strings in fact have more functionality associated with them, because when manipulating text, many more tasks involving character (as opposed to values of arbitrary type) are common and useful. We'll start with the ones familiar from lists. Note that the complete list of methods associated with strings is available in the Python documentation, which describes additional optional parameters not discussed here.

- `<string>.count(<substring>)` returns the number of times `substring` (portion of the string) occurs within the string.
- `<string>.find(<substring>)` returns the index (position) within the string of the first (from the left) occurrence of `substring`. Returns -1 if `substring` cannot be found.
- `<string>.rfind(<substring>)` returns the index within the string of the last (from the left) occurrence of `substring`. Returns -1 if `substring` cannot be found.
- `<string>.index(<substring>)` returns the index within the string of the first (from the left) occurrence of `substring`. Causes an error if `substring` cannot be found.
- `<string>.rindex(<substring>)` returns the index within the string of the last (from the left) occurrence of `substring`. Causes an error if `substring` cannot be found.

```
>>> s = "The quick brown fox jumps slowly over the lazy
   cow"
>>> s.count("ow")
3
>>> s.find("brown")
10
>>> s.find("not here")
-1
>>> s.find("ow")
```

```
12
>>> s.rfind("ow")
48
>>> s.index("ow")
12
>>> s.rindex("ow")
48
>>> s.rindex("not here")
Traceback (most recent call last):
    File "<stdin>", line 1, in ?
    ValueError: substring not found
>>>
```

11.2 Formatting Strings using String Methods

The most commonly used methods on strings are those to change the format of text. With these methods we can change the case of various characters in the text, according to common patterns, pad the text with spaces on the left and right to justify it appropriately or even centre it across a given width, and strip out whitespace in various ways.

- `<string>.capitalize()` returns a copy of the string with only the first character in uppercase.
- `<string>.swapcase()` returns a copy of the string with every character's case inverted (swapped).
- `<string>.center(<width>)` returns a string of width `width` with the original string centred, i.e. the string in the middle of an equal number of spaces on the left and right.
- `<string>.ljust(<width>)` returns the original string left justified (on the left) within a string of width `width`, i.e. padded with spaces up to length `width`.
- `<string>.rjust(<width>)` returns the original string right justified (on the right) within a string of width `width`, i.e. padded on the left with spaces to make a string of length `width`.
- `<string>.lower()` returns a copy of the original string, but with all characters in lowercase.
- `<string>.upper()` returns a copy of the original string, but with all characters in uppercase.
- `<string>.strip()` returns a copy of the string with all whitespace at the beginning and end of the string stripped away.
- `<string>.lstrip()` returns a copy of the string with all whitespace at the beginning of the string stripped away.
- `<string>.rstrip()` returns a copy of the string with all whitespace at the end of the string stripped away.

- `<string>.replace(<old>, <new>)` returns a copy of the string where all occurences of `old` have been replaced with `new`.

```
>>> "a sentence poorly capitalized".capitalize()
'A sentence poorly capitalized'
>>>
>>> "aBcD".swapcase()
'AbCd'
>>>
>>> "center me please".center(60)
'                      center me please
                        '
>>>
>>> "I need some justification here".ljust(60)
'I need some justification here
                                  '
>>>
>>> "No! Real Justification, the RIGHT justification".
   rjust(60)
'             No! Real Justification, the RIGHT
   justification'
>>>
>>> "LOWER me Down".lower()
'lower me down'
>>>
>>> "raise Me UP".upper()
'RAISE ME UP'
>>>
>>> "     I put my whitespace left, I put my whitespace
   right     ".strip()
'I put my whitespace left, I put my whitespace right'
>>>
>>> "Sung to the tune of 'The h0ky p0ky'".replace("0ky",
   "okey")
"Sung to the tune of 'The hokey pokey'"
>>>
>>> "     Losing whitespace on the left.      ".lstrip()
'Losing whitespace on the left.     '
>>>
>>> "      Losing whitespace on the right.     ".rstrip
   ()
'      Losing whitespace on the right.'
>>>
```

11.3 Advanced String Formatting

After all that, let's cut to the chase. The `.format()` function. This allows us to use interpolation (which is explained below) and provides the majority of string formatting operations in a single consistent pattern. Learn it, understand it, appreciate its inner beauty!

This is done using the `.format()` function. Formally put, this interpolates (or inserts) a sequence of values (i.e. a list, tuple, or in some special cases a dictionary) into a string containing interpolation points (placeholders). Wowsers we say? Again in English? The `.format()` function combines a string containing certain codes and a sequence containing values, such that those values are inserted into their respective positions within the string, defined by the position of the codes, formatted according to the specification of those codes, and replacing those codes... Ok, perhaps just an example then...

```
>>> s = "My very {0} monkey jumps swiftly under {1}
   planets".format("energetic", 9)
>>> s
'My very energetic monkey jumps swiftly under 9 planets'
>>>
```

Examining the above example, we had a string containing two '{number}' thingies. The `format` function was called on this, with two parameters. This function served to merge the string at the points where the '{number}' thingies were, with the first parameter replace `{0}`, the second `{1}` and so on.

However, instead of having to put parameters in the exact order you wanted to insert them, you can also use named variables, and let Python figure it out for you.

```
>>> s = "My very {mood} monkey jumps swiftly under {
   number} planets".format(number=5, mood="sad")
>>> s
'My very sad monkey jumps swiftly under 5 planets'
```

Time to get technical. And thingie is not a technical term, except amongst electrical engineers and biochemists. The `.format()` function converts all values in the sequence to strings during the merge. The *placeholder* ('{ }') has a specific format, namely it is contained in '{}' and holds at least one identifier. It's easier to look at the complete format in point form, so here it is...

1. `{ identifier }`
   Placeholders *must* have '{ }' around them with either a parameter number or parameter name inside.

2. `(<mapping/key name>)`
   The parameter identifier may be optionally followed by a key name from a dictionary used as a parameter (i.e. instead of a tuple or list). This is *required* if you use a dictionary to interpolate, as dictionary keys aren't returned in order (and therefore, it doesn't make sense to use numbered parameters to decide where they go.

   ```
   >>> s = "My {0[mood]} monkey jumps swiftly under {0[
      number]} planets".format(
   ```

```
          {"mood": "playful", "number": 3, "something"
             : "else"})
>>> s
'My playful monkey jumps swiftly under 3 planets'
```

3. `{identifier: format}` *optional
   It's also possible to include information on how you want parameters formatted. This is done by including a ':' after the variable name and specifying the format.

   - `#`, `0`, `-`, ` `, `+` *optional
     An optional conversion flag may be used to specify justification and signedness options. Any number of '0', '-', '+', and ' ' can be used in a given conversion specification, and the important ones are:
     - '0': left pad with zeroes, useful for month numbers
     - '-': right pad with spaces, overrides '0' if both given
     - '+': force the use of a plus sign in front of positive numbers
     - ' ': insert a space in front of positive numbers (used to line up with negative numbers where a minus is placed in front.)
   - `<field width and justification>` *optional
     An optional minimum field width. Whatever value is merged in at this point in the string, is converted to a string that is at least as wide as the field width, specified as an integer. Note, because the number is inside a string it must be hard coded, and cannot be an expression. You can also specifiy the justification for the string using one of the following:
     - $<$: left align (default)
     - $>$: right align
     - ^: centre align
   - `.<precision>` *optional
     An optional precision level can be specified (in digits). This will ensure that the precision of floats is shortened to this length. Floats will not be padded.
   - `<conversion type>`
     The conversion type character is a single character specifying the type of value to convert into a string and how the conversion should happen. Some important ones are
     - g: General format. This prints the number as a fixed-point number
     - b: Binary. Prints the number as binary.
     - e: Scientific notation.
     - %: Percentage. Multiplies number by 100 and displays followed by % sign

```
>>> "An integer with field width of three: {0:3}".format
   (5)
```

```
'An integer with field width of three:5   '
>>>
>>> "An integer right justified with a field length of
    3: {0:3>}".format(5)
'An integer right justified with a field length of 3:
    5'
>>>
>>> "An integer with leading zeros: {0:02}".format(5)
'An integer with leading zeros: 005'
>>>
>>> "An integer right justified with forced +: {0:+>}".
    format(5)
'An integer right justified with forced +:  +5'
>>>
>>> "A number with precision 3: {0:.3f}".format(5)
'A number with precision 3: 5.000'
>>>
>>> "A number in scientific notation: {0:e}".format
    (0.0000025)
'A number in scientific notation: 2.5e-06'
>>>
```

11.4 Miscellaneous String Methods

Finally, there are a few miscellaneous methods that prove very useful when dealing with strings. These include

- `<string>.isupper()` return `True` if the string contains only uppercase characters.
- `<string>.islower()` return `True` if the string contains only lowercase characters.
- `<string>.isalpha()` return `True` if the string contains only alphabetic characters.
- `<string>.isalnum()` return `True` if the string contains only alphabetic characters and/or digits.
- `<string>.isdigit()` return `True` if the string contains only digits.
- `<string>.isspace()` return `True` if the string contains only whitespace characters.
- `<string>.endswith(<substring>)` returns `True` if the string ends with `substring`.
- `<string>.startswith(<substring>)` returns `True` if the string starts with `substring`.
- `<string>.join(<sequence>)` returns the elements of `sequence` (which must be strings) concatenated in order with the string between each element.
- `<string>.split([substring])` returns a list of strings, such that the string is split by `substring` and each portion is an element of the returned list. If substring is not specified, the string is split on whitespace.

- `<string>.rsplit([substring])`; the same as split, but the search for the split string is performed from right to left

```
>>> "The quick brown fox".endswith("dog")
False
>>>
>>> "The quick brown fox".endswith("fox")
True
>>>
>>> "The quick brown fox".startswith("A")
False
>>>
>>> "The quick brown fox".startswith("The ")
True
>>>
>>> ", ".join(['1', '2', '3', '4'])
'1, 2, 3, 4'
>>>
>>> "a, b, c, d".split(',')
['a', ' b', ' c', ' d']
>>>
>>> "a, b, c, d".split(', ')
['a', 'b', 'c', 'd']
>>>
>>> "abababa".split("bab")
['a', 'aba']
>>>
>>> "abababa".rsplit("bab")
['aba', 'a']
>>>
```

11.5 Exercises

1. Strings are immutable - the value of a string cannot be modified, but a new string can be created and assigned to the same variable name. How would one thus change a string variable, to for example to insert '-)' after every colon?
2. If we try to use the `.format()` function with a tuple, the tuple must have the same number of elements as there are { } identifiers in the string. Is this the case with dictionaries? Why?
3. Write a program that reads in names until a blank line is encountered, after which it prints out an English style list, i.e. a comma separated list, where the last name is preceded by the word 'and' instead of a comma.
4. What is the value of `"Laziness is a {0}.".format("virtue")`?

5. What is the value of `"{0} days hath {1}, {2}, {3} and {4}. I use my {5} for the other {6}, because I can't remember this rhyme for {7}".format(30, "September", "April", "June", "November", "knuckles", 8, "...")`?
6. What is the value of `"{0:02}/{1:02}/{2:04}".format(10,3,2009)`?
7. What is the value of `"{0:5.3f}".format(3.1415)`?
8. How would you print a column of numbers so the line up right justified for convenient addition?
9. How would you print a user entered string centred in the middle of the console?
10. Write a program that reads in the name, price, and quantity of an item, and stores it in a list of tuples, repeating until a blank product name is entered. It should then print out each item in a nicely formatted manner, using string formatting.
11. Tougher Problem: Modify your answer to question 11 to use products from a dictionary of product codes mapped to product descriptions. Invalid codes should print a warning, and product codes should be integers. The print out at the end should print the full name of the item, followed in brackets by it's code, as well as price and quantity.

Chapter 12

Functions

12.1 Abstracting Common Tasks with Functions

So now we've been programming for nearly two days. You've probably come across a number of tedious tasks that we must 'program repeatedly', as opposed to 'program to repeat'; e.g. looping until a blank line is entered. Often these common tasks are similar, but not exactly the same, e.g. different things need to be done in the loop until a blank line is reached. Ideally we want a labour saving device, and thus we turn to the *function*. We have already been introduced to the use of functions by calling various built in functions, but ultimately we want to be able to make our own functions.

12.2 Defining New Functions

We define a new function using the `def` statement, which takes the form

```
def <funcname>([parameter name][, parameter name][,
    ...]):
     statement
     [statement]
     [...]
```

When this statement is executed, a new object with the name `funcname` is created. Whenever this object is used followed by pair of round braces, the statements listed in the `def` statement, commonly called the *function definition*, are executed, and once completed, execution continues from where it was before the function statements were entered. This process is known as *calling a function*.

It is important to understand that functions provide us with a radically different way to influence the sequence or flow in which our programs execute. Conditions and loops still progress in a roughly top to bottom fashion, but functions allow us to 'dive into' a small block of statements, and come back up to where we were just prior to our dive, more of an in and out form of control.

Of great use is that functions can return a value. In fact in Python, all functions return a value, even if not given a value to return (in which case the return value is `None`). This is done by the use of the `return` statement. The `return` statement, when used within a function definition, simply causes that function to return the value of the expression immediately following the 'return'. Formally, return statements take the form

```
    return <expression>
```

As an example, here is the definition of a simple function to print a triangle of height 3.

```
def triangle3():
    print("*")
    print("**")
    print("***")
```

Henceforth, whenever Python encounters `triangle3()` the three print statements will be executed. Note the brackets! They indicate `triangle3` should be called, rather than treated as a variable. Looking at `triangle3`, there's nothing we couldn't do without cut and paste, so why bother? Well recall that way back in the Basic Input section we said that functions are like mini-programs, in that they take input and produce output. Well that would mean a function could produce different output depending on its input. How do we give a function input?

12.3 Passing and Receiving Parameters by Position

Functions receive their input from *parameters*, which are values given in a comma separated list (i.e. a tuple) when calling the function, and are unpacked within the function automatically. We have to define how many parameters a function takes, and what each ones name is, so we can give a function appropriate parameters later on. For example, let's define a function that accepts one parameter, called `seconds`, and returns how many complete minutes are in that many seconds.

```
>>> def minutes(seconds):
...     print(seconds)
...     return seconds/60
...
>>> minutes
<function minutes at 0xb7cfcdbc>
>>> minutes(71)
71
1
>>> minutes(71.0)
71.0
1.1833333333333333
>>> minutes("bob")
bob
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in minutes
TypeError: unsupported operand type(s) for /: 'str' and
   'int'
>>>
```

In the above example we can already see a number of things about functions and how they handle parameters. Firstly, the `def statement` is executed and `minutes` is created as a function 'object', as evidenced by the value returned after typing in `minutes`. When we call `minutes` by typing `minutes(71)`, Python executes all the statements contained in the function `minutes`. The first statement in the definition prints out the value of seconds as 71. But where did the value 71 come from?

Well, thinking back to tuples, let's treat everything between the open and close brackets in a function call as if it were forming a tuple, so in the case of the function call `minutes(71)` we have formed a tuple `(71, )`. Before the first statement of the function definition is executed, the tuple created by calling the function is unpacked using the names in the function definition's parameter list *in order*. In our case this creates an implicit line before `print(seconds)` which looks like

```
seconds, = 71,
```

Note in the example transcript how the type of the parameter `seconds` changes according to the type of the value used in the function. In fact in the third function call made, the type used actually causes an error, which makes sense since we cannot divide a string by the integer 60. In the error case, we see that the first statement in the function definition (`print(seconds)`) is executed without error, and that the error occurs later. This means we can see the order of execution within a function. More than that, functions are not atomic. This means that if there's an error in your function, all the statements before the error will be executed regardless.

The concepts of parameter passing are best illustrated in the case of multiple parameters. For example, let us define a function which takes two parameters, prints out both and returns the smaller of the two parameters ...

```
>>> def minimum(a, b):
...     print(a)
...     print(b)
...     if a $<$ b:
...         return a
...     else:
...         return b
...
>>> minimum(9, 3)
9
3
3
>>>
```

Okay, we're taking a few extra baby steps here... let's look at what we've done. We've defined a new function called `minimum`, it has two parameters, `a` and `b`. By virtue of putting `a, b` in the parameter list in the function definition, we are implicitly putting the line `a, b = <passed in tuple>` in as the first line, where the passed in tuple is formed by the function call itself, rather than the function definition. Next we notice that we can branch execution flow using an `if` statement within a function definition. We can also loop, in fact we can do pretty much anything we want within a function definition. Finally let's focus on the `return` statement. Each return statement exits the function immediately, and replaces the value of the function call with the value of the expression following the reserved word `return`. So following `minimum(9,3)` we see that program execution 'dives into' the function `minimum`, sets `a` and `b` to 9 and 3 respectively via the implicit tuple unpacking, prints `a`, prints `b`, and never executes `return a` but instead executes `return b`, at which point execution leaves the function definition, and returns to the point of the function call. The value of the function call is thus `3`.

12.4 Composition of Functions in the Definition

Of course, since we can do pretty much anything in a function definition, we can also call other functions, both Python built in functions, and our own defined ones, as in an example to get two coordinates from the user.

```
def getcoords():
    x = input("Enter an X coordinate: ")
    y = input("Enter a Y coordinate: ")
    print("The smaller coordinate is {0}".format(
       minimum(int(x), int(y)) ))
    return x, y
```

So here we have called both a built in function (input) and our own previously defined function (minimum) within the function definition. Also note that we form a tuple in our return statement. We wish to return two values, but the `return` statement can only return the value of a singe expression, so to return multiple values, that expression must be a tuple (or list) of values.

12.5 Composition of Functions in Parameter Lists

Just as expressions can be composed to form more complex expression, so can functions be composed in the strict mathematical sense, i.e. given two functions `f(a)` and `g(b)`, we can obtain a value for f of g of x. Similarly in Python we could say

```
print(minimum(5, minimum(int(input("> ")), 8)))
```

Remember that expressions are evaluated inside to outside, so the value of input is determined first, then it is converted to an integer, then compared against 8 in the inner minimum function, and finally the value of the inner minimum function is compared against 5 in the

outer minimum function. Thus we have composed minimum with itself, and further with input.

In the same way that the range of `g(b)` must fall within the domain of `f(a)`, so too must we ensure in Python when doing our function compositions that the results of inner functions are of appropriate type and value for use as parameters to outer functions.

12.6 Exercises

1. Loops and conditionals both alter the flow of a program. Functions also alter the flow of a program, but in a different way. What is the difference?
2. Write a function that computes the sum of squares of two numbers.
3. Write a function that computes the sum of squares of two or three numbers.
4. Write a function that accepts a list of numbers, and returns the sum of their squares.
5. Write a program that asks the user for a space separated list of data points (i.e. floating point numbers), then uses your function from the previous question to output the sum of squares of those numbers.

Chapter 13

Dictionaries

13.1 What are dictionaries

Dictionaries in Python are known in other languages by other names, including “associative arrays" and “mappings". The latter being perhaps the more explanatory term. Dictionaries are basically a set of mappings from a *key* to a *value*. Keys can be of any non-mutable (not changeable) type, and values can be of any type including other dictionaries, lists, or tuples. Dictionaries specify a set of *key:value* pairs. In this sense, their name is quite descriptive. When thinking of a dictionary of words, each entry contains the word - or key, and it's explanation - or value. Thus, knowing the key allows access to its value. In the case of Python, when the dictionary is accessed using a key it contains, the value associated with that pair is returned.

13.2 Forming a Dictionary

Dictionaries are formed using curly braces (`{ }`), and providing a comma separated list of key:value pairs between the open and close braces.

```
>>> a = {
... 0: "This is the value for key 0",
... 1: "This is the value for key 1",
... "3": "See how keys can also be strings",
... (4, 5): "Or even tuples",
... "and for fun": 7
... }
>>>
```

Here we have a new dictionary, which we have assigned to a variable `a`. The dictionary contains five key:value pairs, each specified using the format `<key> : <value>` within a comma separated list within curly braces. Note how keys can be of multiple different types within the same dictionary. *Key order is not guaranteed by Python.* In other words, it doesn't matter what order you put items into the dictionary, as this order is not maintained. Similarly

values can be of any type, and types can be mixed within one dictionary. Also note the existence of the empty dictionary ...

```
>>> b = {}
>>>
```

13.3 Using Dictionaries

Presumably, the first thing we would want to be able to do after forming a dictionary is to manipulate the key:value pairs. We would want to be able to extract individual pairs by their key name, which we do by using the subscription operator `<dictionary>[<expression>]`

```
>>> a[0]
'This is the value for key 0'
>>> a["3"]
'See how keys can also be strings'
>>> a[4,5]
'Or even tuples'
>>>
```

In a similar way to lists we can change the contents of a dictionary, meaning *dictionaries are mutable*. Specifically, we can change the value of a certain key using the assignment statement `<dictionary>[<expression>] = <expression>`.

```
>>> a["3"] = "Told you keys could also be strings"
>>> a["3"]
'Told you keys could also be strings'
>>>
```

Similarly, a simple assignment statement can add a new key:value pair to a dictionary, as long as the key doesn't already exist in the dictionary. If the key does already exist, it's value is simply changed as in the assignment statement above.

```
>>> a["new"] = "a new pair has been added"
>>> a
{0: 'This is the value for key 0', 1: 'This is the value
    for key 1', '3': 'Told
you keys could also be strings', 'new': 'a new pair has
   been added', (4, 5): 'Or
even tuples'}
>>>
```

It is important to realise that adding a new key:value pair, where the key already exists, means that the value is simply replaced. This means that all key values in a dictionary are unique - that is, no two keys are the same. Also, note that attempting to access a key that doesn't exist causes an error...

```
>>> a[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: 3
>>>
```

Which means that before we can add to the value of a particular key using an assignment such as `a["n"] = a["n"] + 1` we must first ensure the key exists to add to. The same is of course true of any operator applied to a key's value. The reason for this is that assignments evaluate the expression on the right of the equals before assigning it to the variable on the left, meaning the value of `a["n"]` would be looked up before it has been put into the dictionary, causing an error.

Entire dictionaries can be compared using the standard comparison operators. If `a` and `b` are both dictionaries

- `a == b` is `True` (that is, they are seen as equalivalent) only if both `a` and `b` contain exactly the same set of key:value pairs.
- `a is b` is `True` only if `a` and `b` are synonyms for the same dictionary.

This is demonstrated below:

```
>>> a = { "hello": "world" }
>>> b = { "hello": "world" }
>>> a == b
True
>>> a is b
False
>>> c = a
>>> d = c
>>> a == d
True
>>> a is d
True
>>>
```

We can loop over the keys of a dictionary, using a `for` statement.

```
>>> for k in a:
...     print(k)
...
0
1
3
new
(4, 5)
>>>
```

Or perhaps more usefully, we can loop over the keys, and use them as indexes into the dictionary.

```
>>> for k in a:
...     print(k, ":", a[k])
...
0 : This is the value for key 0
1 : This is the value for key 1
3 : Told you keys could also be strings
new : a new pair has been added
(4, 5) : Or even tuples
>>>
```

Finally, we can extract various structures from a dictionary in the form of lists.

- `<dictionary>.values()` returns a view of all values in the dictionary. The order of these values is not guaranteed.

```
>>> a.values()
dict_values([‘This is the value for key 0’, ‘This is
    the value for key 1’, ‘Told you keys
could also be strings’, ‘a new pair has been added’,
    ‘Or even tuples’])
>>>
```

- `<dictionary>.keys()` returns a view containing all keys in the dictionary. The order of these keys is not guaranteed.

```
>>> a.keys()
dict_keys([0, 1, ‘3’, ‘new’, (4, 5)])
>>>
```

- `<dictionary>.items()` returns a view containing tuples of all key:value pairs in the dictionary, where key is the first element of the tuple, and value the second. The order of these pairs is not guaranteed.

```
>>> a.items()
dict_items([(0, ‘This is the value for key 0’), (1,
   ‘This is the value for key 1’), (‘3’,
‘Told you keys could also be strings’), (‘new’, ‘a
   new pair has been added’),
((4, 5), ‘Or even tuples’)])
>>>
```

`a.keys()`, `a.values()` and `a.items()` all return a view of the data. This view is basically a window into the dictionary, that allows you to view parts of it in a specific way. It also means that any changes done to a view, are also performed on the dictionary.

```
>>> k = a.keys()
>>> k
dict_keys([0, 1, ‘3’, ‘new’, (4, 5)])
>>> a["new key!"] = "new value!"
>>> k
```

```
dict_keys([0, 1, '3', 'new key!', 'new', (4, 5))
>>>
```

As you can see in the example above, changing the values in the dictionary `a`, changed the values returned by the view `k`, even though you haven't directly changed `k`.

13.4 Exercises

Given the code:

```
d = {
    "fruits": ["apples", "oranges", "pears", "mangoes"],
    "vegetables": ["tomatoes", "lettuce", "spinach", "
       green peppers"],
    "meat": ["chicken", "fish", "beef", "ostrich"],
    "dairy": ["yogurt", "milk", "cheese", "ice-cream"]
}
```

1. How many keys does `d` have?
2. How many values does `d` have?
3. What is the value of `d["meat"]`?
4. What is the value of `d["dairy"][2]`?
5. How do you access "spinach" using the dictionary `d`?
6. How do you add a new fruit?
7. Consider the set of key:value pairs?
   - "Hitchhiker's Guide to the Galaxy": 1
   - "The Restaurant at the End of the Universe": 2
   - "Life, the Universe, and Everything": 3
   - "So Long, and Thanks for all the Fish!": 4
   - "Mostly Harmless": 5

   (a) How do you create this set as a dictionary in Python?

   (b) How do you find which book in the 'trilogy', i.e. what number, "The Restaurant at the End of the Universe" is?

   (c) Write a program that starts by declaring the above dictionary as a literal, and outputs the books in order.

   (d) Write a program that starts by declaring the above dictionary as a literal, and then asks the user for a number, and prints out name of the book which has the given number.

(e) Write a program the starts by declaring the above dictionary as a literal, then proceeds to switch the keys and values, so that values become keys, and *vice versa*. Print out the resulting dictionary.

8. Write a program that reads in names until a blank line is entered, and prints out each unique name and the number of times it was entered.

9. Write a program the reads strings until a blank line is encountered. For each string entered, treat the portion of the string up to the first colon (or the entire string if no colon is present) as a key name, and everything after the first colon as a value. If the key portion has been entered before, print out the old value paired with that key, and then change the value to the newly entered one. After the blank line, print out a neat list of key value pairs.

10. A small arts and crafts store owner in the middle of the Karoo has recently upgraded to a computerised point of sale system, and wants to do the same for his guest book. Customers have previously left their names a small paragraph of comment in the book. The owner would like his customers to be able to walk up to a computer near the exit, type in their names, and enter a brief comment. He's only interested in a customer's most recent comments, and doesn't want store old comments. So repeat customer's must be able to update their previous comments. When a repeat customer types in their name, their previous comment is displayed back to them, and they can enter a new comment. Should they enter a blank line instead of a comment, their previous comment is preserved. Also, if instead of a customer name the special command 'quit' is entered, the program exits. Similarly the command 'showcomments' causes all customers' names to be displayed, followed by their comments slightly indented. Customer's must be able to enter their names in a case insensitive manner.

11. Extend your solution to the previous problem, by allowing customers to enter multi-line comments, and to terminate their comments by entering a blank line. If the comment is entirely blank, i.e. the first line is blank, then it does not overwrite the former comment if any. Also, ensure that when the comments are outputted back, either because of the 'showcomments' command, or a repeat customer entering their name, that the line width of the outputted comments does not exceed 60 characters, nor break a word in two, i.e. lines are only broken on white space.

12. What happens if you make an element of a dictionary point to itself?

Chapter 14

Scope

14.1 The Concept of Scope

The concept of scope refers to the idea that variables have a limited availability according to the block of statements in which they were originally defined. We could say that the structure of blocks of statements in a Python program resembles a set of bulleted lists lying variously within one another. On the outside we have our number list (our 'main block'), and all other lists (blocks), are contained in it, either directly or sub-contained within another inner block.**The scope of a variable is the set of blocks within which it is available, i.e. defined.**

The outermost block, i.e. our main program, or everything beginning in column 1 of our code, is generally referred to as **the global scope**.

14.2 Pythons Golden Exception err... Rule

Python in general sets the scope of a variable to be the block in which it is defined and consequently all blocks inside of that block, to any arbitrary level. This means that if some variable 'a' has been defined in a block, and I wish to use it's value in that block or any other block inside of that block, I can. 'a' will be there.

One analogy is that of the program as a large open room. Variables are slips of paper on which values are written on one side, and the variables' names on the other. Whenever we need the value of a variable, for example when a variable is referenced in an expression, we can quickly scan through the papers in the room for the one with the right name, and read the value off the other side. But when we call a function, it's like putting up a wall of one way glass that divides the room in half, behind us as we walk into the function's area. We can look back through the glass, and still see all the papers that were in the room originally, and thus their names and values, but someone standing on the other side can't see any papers we may use inside the function enclosure, i.e. variables assigned values in the function. When we exit the function, and take down the glass wall in doing so, we also destroy all the papers used on the inside of the wall.

More formally speaking, when a name (i.e. not a reserved word, number, string or symbol) is encountered Python looks for the most recent definition of the variable, object, or function that has that name in the current block (i.e. the block in which execution is currently happening), and failing finding a definition in the current block will search outwardly in a block wise fashion to the outermost block.

Note that the fact that the **most recent** definition is searched for means that if an assignment is made (essentially a re-definition) in the current block (**known as the local block**) overrides a definition or value assigned in an outer block, **even before the assignment is encountered**.

```
#!/usr/bin/python
#A script to illustrate variable scope in Python

total_height = 0

def triangle():
    def inc_total_height():
        total_height = total_height + height

    for i in range(height):
        print("*" * i)
    inc_total_height()
    print()

def square():
    def inc_total_height():
        total_height = total_height + height

    for i in range(height):
        print("*" * height)
    inc_total_height())
    print()

height = 3
triangle()
height = 4
square()
height = 5
triangle()
```

In the example above we are looking at the variables and their respective scope. The scope of a variable is indicated by its colour. Variables of a particular colour are available only in the block of that colour, and that block's inner blocks. The two inner blue blocks are separate from one another, as are their respective inner brown blocks. In the brown blocks, when the variable `height` is encountered, Python tries to find the value of `height` within the brown block. When it can't find it, it tries the next block outwards, namely the blue block, and

failing there, finally tries the last block (the gray one), where it finds height. If Python fails to find a value in the outermost block, it cause an error, stating that the variable has not been defined. Therefore, `height`, for example, is gray, because it was defined in the gray block by it's initial assignment. Similarly `i` is blue, and **unavailable** in the gray block, and the other blue block. `total_height` is gray, except in the brown blocks, where it occurs in gray and brown, but why? The assignment statement in the brown blocks causes `total_height` to be redefined (that is, created again), so Python knows to look for its value only in the brown block in question, and when it can't find it there, causes an error (`total_height` not defined) because it assume you are try to access the value of `total_height` before assigning it an initial value. In fact, the fact that there is an assignment in the brown block causes `total_height` to become a totally new variable unrelated to the original one, which only exists in the brown block. This effect is consistent (the same) throughout the brown block, even before the assignment statement has been executed.

14.3 Forcing Global Scope for a Particular Variable

To override Python's redefinition effect applied to `total_height` we can tell Python explicitly to use the `total_height` variable from outside the local block (that is, use the value from the gray block, rather than the brown one we are now in). We do this by declaring that variable name to be **global**, using the `global` statement, which takes the form `global <variable name>`. The `global` statement tells Python to use the innermost variable of the given name, instead of creating a new locally available variable. So let's change our program slightly so does what we expect, i.e. doesn't break, and counts the total number of lines outputted as a square or triangle.

```
#!/usr/bin/python
#A script to illustrate variable scope in Python

total_height = 0

def triangle():
    def inc_total_height():
        global total_height
        total_height = total_height + height

    for i in range(height):
         print("*" * i)
    inc_total_height()
    print()

def square():
    def inc_total_height():
        global total_height
        total_height = total_height + height

    for i in range(height):
```

```
        print("*" * height)
    inc_total_height())
    print()

height = 3
triangle()
height = 4
square()
height = 5
triangle()
```

Chapter 15

Importing modules

15.1 The `import` Statement

We have already learnt the basics of programming. We have all the tools we will ever need to solve any problem we can solve in our heads. But many times we will find ourselves reinventing the wheel. Either a wheel we've made ourselves, e.g. writing the same piece of code over and over again in different programs, or even a wheel someone else has made, e.g. solving a problem someone else has already solved. The point is the code has already been written, and what we want is a convenient way to package it so that we can use it without rewriting it. And thus the *module* was born.

Modules allow us to write code, and separate it out from other code into a different file, known as a 'module'. We can use the `import` statement to include the code of another file, either in the current directory, or in python's *search path*, in place. What this means is that the file is loaded and run as if its code had been in the place of the `import` statement. As such, any statements in the file are executed, including function definitions, class definitions, assignments, etc.

```
import <module_name>
```

`module_name` is the name of a Python file without the '.py' extension. Any Python program can be imported as a module, although most often modules contain function definitions and a minimum of initialisation code, allowing us to keep commonly used function definitions in a place where they can be reused without rewriting them, and more importantly collecting function definitions that are related, e.g. all database functions, together and separate from our main code.

15.2 Namespaces

If we have a Python file which we wish to import, that has a global variable `i`, into our main program, also with a global variable `i`, there could be some confusion. The module's `i` variable is in the global scope of the module, but where is it in the scope of the program

as a whole? Python solves this problem with the introduction of namespaces. Formally, a module when imported introduces its own scope block, called the *module scope. It is a global scope in its own right.* Variables within a modules global scope are forcibly kept separate from their super-programs, or super-modules (i.e. those programs or modules that import them). The global statement does not allow variables references or assignment to cross module scope boundaries. Instead all names (variables, functions, or classes) are created within a module's 'namespace', meaning they must be explicitly referenced from without the module by first naming the module itself, as in

```
<module_name>.<symbol_name>
```

Where `symbol_name` is the name of a variable assigned a value, a function defined, or of a class defined.

```
#my_module.py

i = 2
```

```
#main.py
i = 1

import my_module.py
print i
print my_module.i

print "---"

my_module.i = 3
print i
print my_module.i
```

Running main.py produces the following output

```
1
2
---
1
3
```

Despite the fact that `i` is assigned the value of 2 in my_module.py *after* it is assigned the value of 1 in main.py, since the `import` statement executes that assignment after the initial assignment, the value of `i` in main.py is not changed. As we can see, the two `i`'s are kept completely separate. From main.py, if we wish to reference the `i` in my_module.py, we must do explicitly, using the module name, `my_module.i`. We also see that we can assign a value to a variable in a module's namespace, using the same syntax.

15.3 The `dir` function

```
dir(<object>)
```

The `dir` function is a built in function that takes one parameter, and lists all the attributes of the object given. Methods, properties, and variables are all considered attributes. If this isn't making sense to you yet, don't worry, we will cover it all in the sections on object oriented programming. What we need to know now, is that although the `dir` function is not very helpful in a program, it is *incredibly useful* when using the Python interactive interpreter. As modules are objects in python, in fact everything is an object, we can call `dir` on a module to obtain a list of everything in the module that can be referenced with the dot notation (`<module_name>.<attribute>`).

```
>>> import my_module
>>> dir(my_module)
['__builtins__', '__doc__', '__file__', '__name__', 'i']
>>>
```

In the example, we import my_module.py (from above), and run `dir` on it. We are returned a list containing some stuff we expect, namely the `i`, which is a variable assigned a value within the module my_module, and some stuff we don't expect; `['__builtins__', '__doc__', '__file__', '__name__']`. Short of delving into both object oriented programming and exception handling all at once, these elements cannot be easily explained now, and are not especially important to us.

15.4 The `locals` and `globals` Functions

Of more direct use to use whilst programming, are the `locals` and `globals` functions. Both take no parameters, and both return dictionaries. `globals()` returns a dictionary of objects available in the global scope, whilst `locals()` does the same but for the local scope. The practical use of this is we can make even the names of our variables dynamic, i.e. the ability to use a variable (`a`) to store another variable's (`b`) name, by using `a` as a key into the dictionary returned by either locals, or globals.

```
#!/usr/bin/python

#ask the user for a variable name
vname = input("Enter a variable name: ")

#ask the user for a value
value = input("Enter a value for %s: "%vname)
locals()[vname] = value

print "vname: ", vname
print "locals()[vname]: ", locals()[vname]
print "value: ", value

#assume the user inputted bob
print "bob: ", bob
```

Outputs the following when 'bob' is entered as a variable name, and 4 as a value. Note how 'bob' becomes available as a normal variable name, as seen in the last line, once it has been added to the `locals()` dictionary.

```
Enter a variable name: bob
Enter a value for bob: 4
vname:  bob
locals()[vname]:        4
value:  4
bob: 4
```

15.5 An Overview of Selected Standard Modules

Ultimately, the true use of modules, is to get other people to do the work for us. We are programmers, and thus laziness a virtue, after all. The comprehensive list of standard Python modules, i.e. modules that come with a standard install of the latest version, can be found at http://docs.python.org/modindex.html. Of particular interest to us, by virtue of their usage most common, are

math Provides various non-basic mathematical functions, *inter alia*: trigonometric functions, hyperbolic functions, exponent and logarithm functions, and mathematical constants (e.g. pi and e)

random Provides various methods of choosing random numbers, or making random choices from sequences, using a variety of statistical distributions.

os Provides multitudinous operating system related functions, primarily file and process management functions.

sys Provides a large number of variables and functions dealing with the internals of the Python interpreter itself, and the environment in which it was invoked. Specific uses of various sys variables are discussed shortly.

time Provides various functions for obtaining the current time, and manipulating variables that store values representing time.

15.6 The sys Module

The `sys` module is quite extensive, but by far the most useful aspects of it are access to what are known as the *standard streams*, and the processes command line arguments. When a program is run from the command line, for example Python, we can pass the program itself parameters, as if it were a function. In operating system shell speak they are known as arguments. We pass an argument to Python to tell it what Python file to run,

```
sirlark@hephaestus ~ $ python my_file.py
```

We can actually pass any number of arguments to a program we run from the command line, each separated by some amount of white space. If white space is enclosed in quotes of

some kind, it does not separate arguments, but rather is included within one command line argument. In Python's case, arguments that come after the specification of the filename to run are passed through to the program being run as its own command line arguments.

```
{sirlark@hephaestus ~ $ python my_prog.py somefile.fasta
    7 "ACCTGT AGTCA"
```

In the example above, the program my_prog.py receives three command line arguments, namely `somefile.fasta`, `7`, and `ACCTGT AGTCA`. Note the space within the sequence snippet, and the fact that the 7 is in quotes and thus a string.

So now we have a really convenient way of passing small amounts of input into our programs, instead of prompting the user with input calls all the time. Command line arguments are especially useful for the purposes of specifying filenames and options that influence how our program processes data, and preferable to input statements for two reasons. First, when entering filenames, command line arguments allow the user to use shell expansions and tab completions. Second, our program can be run without user intervention if it's likely to take a long time to run, e.g. the user can specify a number of files to process on the command line (probably using a '*.fasta' or similar) and walk away. Assuming each file would take 20 or more minutes to process, if we were using input statements to prompt for the next file, if the user came back the next morning, our program would have processed the first file, and be waiting for the entry of the second filename. Command line arguments avoid this issue neatly.

But how, in Python, do we access what command line arguments have been given to our program? Well, the `sys` module provides a list called 'argv' (for argument vector), which contains as its elements the command line arguments passed to our program, in order. Examine the following simple program. Run it a few times providing different command line arguments each time, and test it with quoted arguments containing spaces or tabs.

```
#!/usr/bin/python
#commargs.py

#import the sys module
import sys

#print out our command line arguments
print sys.argv
```

```
nyx@nyx-desktop:~/umonya_notes$ python commargs.py Hello
['commargs.py', 'Hello']
nyx@nyx-desktop:~/umonya_notes$ python commargs.py Hello
    There
['commargs.py', 'Hello', 'There']
nyx@nyx-desktop:~/umonya_notes$ python commargs.py "
   Hello There"
['commargs.py', 'Hello There']
nyx@nyx-desktop:~/umonya_notes$ python commargs.py "
   Hello There" 47
```

```
['commargs.py', 'Hello There', '47']
nyx@nyx-desktop:~/umonya_notes$
```

Note how the first element of `sys.argv` is always the name of our Python program. Where after the arguments are those we gave on the command line. Also note how all the elements, even the 47, are in fact strings.

15.7 The Standard Streams

Whenever a Python program is run, it opens three files automatically, called standard input (`stdin`), standard output (`stdout`), and standard error (`stderr`). Generally these files represent keyboard input, screen output, and error output respectively, but they can be *redirected* causing the input to our program to come from the output of another, for example, as when used in a shell pipe ('|'). Similarly our programs output could be redirected to a file, or piped into the input of another process. However, all these effects are transparent to us... the `print` statement actually writes to `stdout`, the input function reads from `stdin`. If we want to write to `stderr`, however, we must do so explicitly. The three standard streams are all accessible via the `sys` module as

- `sys.stdin`: standard input
- `sys.stdout`: standard output
- `sys.stderr`: standard error. This stream exists to differentiate error output from normal output, so that we can use the shell to split off errors from processes executed in large batches, and not have to deal with all the normal output they produce. This stream is usually outputted to the screen, unless it has been redirected, which means that programs can generally issue error messages to the screen even if their normal output has been redirected.

Each of these are file objects, so `sys.stdin`, which is opened only for reading, has the usual `.read`, `.readline`, and `.readlines` methods, whilst the other two, `stdout` and `stderr`, being opened only for writing have the `.write` and `.writelines` methods available. Here's an example of how to use `stdout` and `stderr` to split output effectively.

```
#!/usr/bin/python

import sys

#get a line of input from the user
print "Please enter a phrase: " #this gets sent to
   stdout
phrase = sys.stdin.readline()   #this gets read from
   stdin

if phrase.isdigits():
    sys.stderr.write("Phrase entered was a number\n")
       #this gets sent to stderr
```

```
else:
    sys.stdout.write("Phrase contains %i 'a' characters\
       n"%phrase.count('a'))
```

There's one more feature of the `print` statement that now becomes important, namely redirection. We can specify a standard stream (or in fact any file object open for writing) to print to, instead of standard output. The syntax is:

```
print >> <file object>, <expression>[, <expression>...]
```

So the above example could be redone as follows:

```
#!/usr/bin/python

import sys

#get a line of input from the user
print "Please enter a phrase: " #this gets sent to
   stdout
phrase = sys.stdin.readline()   #this gets read from
   stdin

if phrase.isdigits():
    print >> sys.stderr, "Phrase entered was a number"
       #this gets sent to stderr
else:
    print "Phrase contains %i 'a' characters"%phrase.
       count('a'))
```

15.8 Exercises

1. What is redirection, and what are it's effects?
2. When, and why, would you want to output to `stderr`?
3. When, and why, are command line arguments preferable to taking input from the keyboard?
4. Can you think of uses for `sys.argv[0]`? What are they?

Chapter 16

Understanding errors

16.1 Traceback most recent what?!?

We are by now all familiar with the dreaded `Traceback (most recent call last):`. The sign that something has gone wrong, that your late night just got later, your assignment is rapidly becoming overdue, and that there's not nearly enough coffee left in the jug! Well, believe it or not, the messages Python prints are there to help you, as long as you are willing to read and understand them.

16.2 The Call Stack

First we need to understand the concept of the *call stack*. When we are running our Python program, the execution point moves around, from top to bottom, branching at `if` statements, looping with `while` or `for` statements, or jumping inside functions... But this causes issues. If an error occurs in our main program block, it is easy to find the problem by looking at the line on which the error occurred. There is generally only one way for execution to reach that line. But in a function, which can be called from many places within our program, including itself or other functions, knowing that the error occurred on a line in a particular function definition doesn't help us understand how the flow of execution led up to the error.

Enter the *call stack*! Every time a function is called, Python pushes it on to something called the *call stack*. Think of a stack like getting into the elevator in a crowded mall. You get in first, and as more people enter the elevator, you are forced to the back. When you arrive at your floor, the last person in, now being closest to the door, exits first. If you want to get out in the middle somewhere, everyone who got in after you must briefly step out, to let you out, then go back in. This is exactly what happens with the *call stack*. As functions are called, their names are placed on the stack, and as they finish, their names are removed.

The *traceback* presents us with the list of called function, from the first called to the most recent called (`most recent call last`). It gives us the following information:

- the file where the call occurred,

- the line in that file,
- the name of the function the call was made from (or '?' if it isn't a function), and
- the line where the next function was called, or where the error occurred.

So let's go through the following traceback:

```
Traceback (most recent call last):
  File "test.py", line 25, in ?
    triangle()
  File "test.py", line 12, in triangle
    inc_total_height()
  File "test.py", line 8, in inc_total_height
    total_height = total_height + height
UnboundLocalError: local variable 'total_height'
   referenced before assignment
```

We see that execution started in the file test.py and proceeded to line 25, where the function `triangle` was called. Within the function `triangle`, execution proceeded until line 12, where the function `inc_total_height` was called. Within `inc_total_height` and error occurred on line 8.

16.3 So what went wrong?

The last line, or sometimes two lines, of the traceback describe what actually went wrong. In Python, all errors are given a class, e.g. input/output error, undefined variable or index out of range. These classes are given names that are one word without spaces. The class name of the error describes roughly what type of error occurred and it is printed first. Then everything following the colon (':') is further description of the actual variables, expressions, and statements involved in the error, and what they were doing. So we can tell from the above example that the class was `UnboundLocalError` and that the local variable `total_height` was referenced, or evaluated, before a value was assigned to it.

Learning the meaning of the class names comes with time, but the description following is usually sufficient to point you in the right direction. At least now you can track down exactly what line is causing the problem, and how the program got there.

16.4 Error types

As mentioned before, Python errors are divided into classes - lots of them! Luckily only a handful are very common. These are listed here, in alphabetical order, so that you can figure out what you did wrong when you encounter one of these errors. Remember that there are various things that cause these errors. Only the most common reasons are mentioned here. Copy and paste the error message right into a Google search and you are sure to find an answer. And don't worry - everyone, even experienced software engineers, run into these on a regular basis.

IndentationError

```
for i in range(0, 10):
...     b = i
...   print(i)
  File "", line 3
    print i
          ^
IndentationError: unindent does not match any outer
   indentation level
```

IndentationError is actually a special type of SyntaxError. You get this error when your indentation is not consistent. In the example `b = i` has 4 leading spaces but `print(i)` only has 2. If `print(i)` needs to be executed for every iteration of the `for` loop, then it needs to have the same level of indentation as the inner block, i.e. the same as `b = i`. Otherwise, if it is only meant to execute after the `for` loop, it needs to have 0 leading spaces.

NameError

```
print(variable)
Traceback (most recent call last):
  File "", line 1, in
NameError: name 'variable' is not defined
```

A `NameError` occurs when you try to use a variable which does not exist in the current *scope*. In other words it has not been defined in the scope you are trying to use it in. Perhaps you haven't defined it at all, or perhaps you misspelt the variable name.

IndexError

```
i = [‘hello’, ‘world’]
i[2]
Traceback (most recent call last):
  File "", line 1, in
IndexError: list index out of range
```

This error is an easy one to understand. In the example the list `i` only has 2 elements. Those elements have index 0 and 1 (or -2 and -1 if you use reverse indices). Thus index 2 is *out of range* because it is too large - there is no value in the list at this index. `IndexError` will happen for any index that is too large or too small.

SyntaxError

```
print(‘hello world!)
  File "", line 1
    print(‘hello world!)
                       ^
SyntaxError: EOL while scanning string literal
```

```
answer = ((1 + 3) * 5 / 6.0 ;
  File "", line 1
    answer = ((1 + 3) * 5 / 6.0 ;
                                ^
SyntaxError: invalid syntax
```

Syntax errors occur when Python cannot understand your code because you didn't use the correct 'grammar'. While you can get away with the occasional grammar mistake in English, the same cannot be said of computer languages. Things have to be exactly right. Luckily syntax errors are easy to solve if you understand the error message. The '^' character in the error message points to the place where Python encountered the error. In the first example above, a closing single quote is missing. In the second example it is a closing bracket that is missing. Always check that you having matching closing and opening brackets and quotes.

```
File "Desktop/errors.py", line 1
    for i in range(0, 10)
                         ^
SyntaxError: invalid syntax
```

Another common syntax error is leaving out the colon (':') at the start of a code block. `if`, `for` and `while` statements, as well as function definitions, all start new code blocks. So those lines have to end with a colon.

TypeError

```
'ab' * 'ab'
Traceback (most recent call last):
  File "", line 1, in
TypeError: can't multiply sequence by non-int of type '
   str'
```

```
def multiply(a, b):
    return a * b

multiply(10)
Traceback (most recent call last):
  File "", line 1, in
TypeError: multiply() takes exactly 2 arguments (1 given
   )
```

When you use the wrong type of variable for something in Python, you will get a TypeError. In the first example, Python cannot multiply two variables that are strings. At least one of them needs to be a number. The second example is an altogether different `TypeError`. The function `def multiply(a, b)` needs two arguments to be passed to it, e.g. `multiply(10, 15)`. Too many or too few arguments will also cause a TypeError.

ValueError

```
int(‘abc’)
Traceback (most recent call last):
  File "", line 1, in
ValueError: invalid literal for int() with base 10: ‘abc
    ’
```

Value errors can occur for many different reasons. It means that there is something ‘wrong’ about the value of a variable or literal. In the example above the value in question is `‘abc’` and the program is trying to turn `‘abc’` into an integer. How does one turn `‘abc’` into a number? I don’t know and neither does Python. That is why a `ValueError` pops up.

Remember, if you cannot find an answer here, just copy-paste the error message into Google!