tutorialv3

Alan Gauld

12 April 2010

tutorialv3

Table of Contents

Contents

Learning to Program

by Alan Gauld

Introduction - What, Why, Who etc.

Some Background

The reason I created this tutorial originally is that two friends wanted to learn how to program and asked for my help. I was amazed to discover that while there were many programming web sites and tutorials on the web there was virtually nothing that taught programming to complete beginners. So I wrote one. That situation has changed and there are now many sites for beginners and I provide links to some of them at the bottom of this page. However my approach is still unique and as such may appeal to some learners more than the other sites, so here it stays.

The idea of learning to program is still, I believe, a good one for most computer users, even if they do not ever write any significant programs themselves. Understanding how programmers think can help make applications more logical and user friendly. Also many applications allow customisation by writing little programs known as macros. And of course there is the web with the opportunity to publish your own web site and sooner or later you will want to add some dynamic features to your web pages, and that means programming. Finally the Internet and the Web encourage a general interest in computers and that interest naturally leads to a desire to "take control", which means learning to program!

Why me? Well I am a professional programmer who came to programming from an electronic engineering background. I have used (and continue to use) several computer languages and don't have any personal interest in promoting any particular tool or language.

What will I cover

As much as I can. I will cover the basic theory of computer programming - what it is, some of its history and the basic techniques needed to solve problems. I will not be teaching esoteric techniques or the details of any particular programming language, in fact I'll be using several different languages, since I believe its important to realize that different languages do different things well. That said, the majority of the course will be in the language called Python.

Who should read it?

Put another way: what do I expect the reader to know already?

I expect the reader of this tutorial to be an experienced user of a computer system, probably running Windows, MacOS or Linux although others should be able to cope too. I also expect them to understand some very basic mathematical concepts such as how to calculate areas of simple shapes, geometric coordinates,

sets, and basic algebra. These are all important in todays programming environments, and many programming concepts are based on these ideas. However the depth of knowledge needed is very low and if you do find the math getting too hard, you can usually just skip over a few paragraphs, try the code as it is and hopefully the penny will drop even if the math still confuses you.

One thing you should know is how to run commands from your operating system's command prompt. In Windows this is variously known as a *DOS box*, the *MS DOS Window* or *MS-DOS Prompt*, or occasionally, nowadays, the *CMD Box*. Basically it's a black window with a white text prompt that usually says `C:\WINDOWS>` and you can start it by going to the `Start->Run` dialog and typing `CMD` into the entry box and hitting OK. If you use Linux then you should know all about terminal windows and on MacOS you can run the *Terminal* program under Mac OS X (which is found in the `Applications->Utilities` folder). There are lots of powerful shortcuts that can save you typing time if you care to read the help files for your Operating System prompt. I won't cover those here. One tutorial for Windows users can be found here. And a basic Unix shell primer can be found here.

I will not be covering issues like how to create or copy text files, how to install software, or the organization of files on a computer storage system. Frankly if you need to know those things you probably are not at the stage of being ready to program, regardless of your desire to do so. Find a tutorial for your computer first, then when you're confident with the above concepts revisit this site. Remember that Windows and MacOS both have comprehensive help systems built in. Linux has a huge amount of tutorial material on the web, Google is your friend...

Why Python?

Python happens to be a nice language to learn. Its syntax is simple and it has some very powerful features built into the language. It supports lots of programming styles from the very simple through to state of the art Object Oriented and Functional techniques. It runs on lots of platforms - Unix/Linux, MS Windows, Macintosh etc. It also has a very friendly and helpful user community. All of these are important features for a beginner's language.

Python however is not *just* a beginner's language. As your experience grows you can keep on using Python either as an end in itself or as a rapid prototyping language. There are a few things that Python is not well suited to, but these are comparatively few and far between.

I will also use VBScript and JavaScript as alternatives. The reason for this is to show that the same basic techniques apply regardless of the language details. Once you can program in one language you can easily pick up a new one in a few days. Why those languages? Well, for a start they have very different styles to Python so form a useful contrast, and more prosaically if we accept that most Web surfers who are also beginners are using PCs with Microsoft Windows installed, there is a programming environment built in to the operating system called Windows Scripting Host which has support for VBScript and JScript (which is Microsoft's variant of JavaScript). In addition, anyone using Microsoft's web browser can also use these languages within their browser and, in fact,

Or in PDF format.

Send Feedback

JavaScript should work in almost any browser on any Operating System. Initially we'll only look at how to run VBScript and JavaScript inside a browser, but I will be introducing WSH in some of the later topics as an optional extra.

Other resources

There are other Web sites trying to do this in other languages (and in the time since I originally created this site a few other Python sites have appeared). There are also lots of tutorials for those who already know how to program but want to learn a new language. This section contains links to some of those that I think are interesting!

- The official Python language web site with online documentation, latest downloads etc.
- The official Perl web site - Perl is a natural competitor to Python in capability but is, I think, harder to learn.
- JavaScript. is the source for information about JavaScript.
- If you don't much like my style a web site with similar aims is the How to think like a Computer Scientist produced by Jeff Elkner who uses Python in his high School classes. It seems a little bit less comprehensive than mine, but maybe I'm just biased :-)
- Finally, if you are an adventurous sort with a good math background you might try the How to Design Programs web site which is also available as a paper book. It teaches a dialect of the Lisp programming language called Scheme. It is very effective at introducing a methodical approach to building programs.
- Since I first wrote this tutor a whole bunch of non programmer's tutorials have appeared and they are listed on the Python web site, so you can take your pick. Most of them focus on just getting you programming in Python so they don't explain so much of the jargon as I do, nor do they explain the Computer Science theory like Jeff does. You can find the page here.

Next Contents

If you have any ideas on how to improve this tutorial
please feel free to email me

Learning to Program

by Alan Gauld

Introduction - What, Why, Who etc.

Some Background

The reason I created this tutorial originally is that two friends wanted to learn how to program and asked for my help. I was amazed to discover that while there were many programming web sites and tutorials on the web there was virtually nothing that taught programming to complete beginners. So I wrote one. That situation has changed and there are now many sites for beginners and I provide links to some of them at the bottom of this page. However my approach is still unique and as such may appeal to some learners more than the other sites, so here it stays.

The idea of learning to program is still, I believe, a good one for most computer users, even if they do not ever write any significant programs themselves. Understanding how programmers think can help make applications more logical and user friendly. Also many applications allow customisation by writing little programs known as macros. And of course there is the web with the opportunity to publish your own web site and sooner or later you will want to add some dynamic features to your web pages, and that means programming. Finally the Internet and the Web encourage a general interest in computers and that interest naturally leads to a desire to "take control", which means learning to program!

Why me? Well I am a professional programmer who came to programming from an electronic engineering background. I have used (and continue to use) several computer languages and don't have any personal interest in promoting any particular tool or language.

What will I cover

As much as I can. I will cover the basic theory of computer programming - what it is, some of its history and the basic techniques needed to solve problems. I will not be teaching esoteric techniques or the details of any particular programming language, in fact I'll be using several different languages, since I believe its important to realize that different languages do different things well. That said, the majority of the course will be in the language called Python.

Who should read it?

Put another way: what do I expect the reader to know already?

I expect the reader of this tutorial to be an experienced user of a computer system, probably running Windows, MacOS or Linux although others should be able to cope too. I also expect them to understand some very basic mathematical concepts such as how to calculate areas of simple shapes, geometric coordinates, sets, and basic algebra. These are all important in todays programming environments, and many programming concepts are based on these ideas. However the depth of knowledge needed is very low and if you do find the math getting too hard, you can usually just skip over a few paragraphs, try the code as it is and hopefully the penny will drop even if the math still confuses you.

One thing you should know is how to run commands from your operating system's command prompt. In Windows this is variously known as a *DOS box*, the *MS DOS Window* or *MS-DOS Prompt*, or occasionally, nowadays, the *CMD Box*. Basically it's a black window with a white text prompt that usually says `C:\WINDOWS>` and you can start it by going to the `Start->Run` dialog and typing `CMD` into the entry box and hitting OK. If you use Linux then you should know all about terminal windows and on MacOS you can run the *Terminal* program under Mac OS X (which is found in the `Applications->Utilities` folder). There are lots of powerful shortcuts that can save you typing time if you care to read the help files for your Operating System prompt. I won't cover those here. One tutorial for Windows users can be found here. And a basic Unix shell primer can be found here.

I will not be covering issues like how to create or copy text files, how to install software, or the organization of files on a computer storage system. Frankly if you need to know those things you probably are not at the stage of being ready to program, regardless of your desire to do so. Find a tutorial for your computer first, then when you're confident with the above concepts revisit this site. Remember that Windows and MacOS both have comprehensive help systems built in. Linux has a huge amount of tutorial material on the web, Google is your friend...

Why Python?

Python happens to be a nice language to learn. Its syntax is simple and it has some very powerful features built into the language. It supports lots of programming styles from the very simple through to state of the art Object Oriented and Functional techniques. It runs on lots of platforms - Unix/Linux, MS Windows, Macintosh etc. It also has a very friendly and helpful user community. All of these are important features for a beginner's language.

Python however is not *just* a beginner's language. As your experience grows you can keep on using Python either as an end in itself or as a rapid prototyping language. There are a few things that Python is not well suited to, but these are comparatively few and far between.

I will also use VBScript and JavaScript as alternatives. The reason for this is to show that the same basic techniques apply regardless of the language details. Once you can program in one language you can easily pick up a new one in a few days. Why those languages? Well, for a start they have very different styles to Python so form a useful contrast, and more prosaically if we accept that most Web surfers who are also beginners are using PCs with Microsoft Windows installed, there is a programming environment built in to the operating system called Windows Scripting Host which has support for VBScript and JScript (which is Microsoft's variant of JavaScript). In addition, anyone using Microsoft's web browser can also use these languages within their browser and, in fact, JavaScript should work in almost any browser on any Operating System. Initially we'll only look at how to run VBScript and JavaScript inside a browser, but I will be introducing WSH in some of the later topics as an optional extra.

Other resources

There are other Web sites trying to do this in other languages (and in the time since I originally created this site a few other Python sites have appeared). There are also lots of tutorials for those who already know how to program but want to learn a new language. This section contains links to some of those that I think are interesting!

- The official Python language web site with online documentation, latest downloads etc.
- The official Perl web site - Perl is a natural competitor to Python in capability but is, I think, harder to learn.
- JavaScript. is the source for information about JavaScript.

- If you don't much like my style a web site with similar aims is the How to think like a Computer Scientist produced by Jeff Elkner who uses Python in his high School classes. It seems a little bit less comprehensive than mine, but maybe I'm just biased :-)
- Finally, if you are an adventurous sort with a good math background you might try the How to Design Programs web site which is also available as a paper book. It teaches a dialect of the Lisp programming language called Scheme. It is very effective at introducing a methodical approach to building programs.
- Since I first wrote this tutor a whole bunch of non programmer's tutorials have appeared and they are listed on the Python web site, so you can take your pick. Most of them focus on just getting you programming in Python so they don't explain so much of the jargon as I do, nor do they explain the Computer Science theory like Jeff does. You can find the page here.

Next Contents

If you have any ideas on how to improve this tutorial
please feel free to email me

What do I need?

What will we cover?
The character and mindset of a programmer, the programming environments used in the tutor.

Generally

In principle you don't *need* anything to do this course other than an Internet enabled computer - which I assume you have if you are reading this in the first place! The other thing that is useful is the right *mind set* to program. What I mean by that is an innate curiosity about things, coupled to a logical way of thinking. These are both essential requirements for a successful programmer.

The curiosity factor comes into play in looking for answers to problems and being willing to dig around in sometimes obscure documents for ideas and information needed to complete a task.

The logical thinking comes into play because computers are intrinsically stupid. They can't really do anything except add single digits together and move bytes from one place to another. Luckily for us some talented programmers have written lots of programs to hide this basic stupidity. But of course as a programmer you may well get into a new situation where you have to face that stupidity in its raw state. At that point you have to think for the computer. You have to figure out exactly what needs to be done to your data and when.

So much for the philosophy! However if you want to get the best from the tutorial you will want to follow along, either typing in the examples by hand or cutting and pasting from the Web page into your text editor. Then you can run the programs and see the results. To do that you will need to have Python installed on your system (and for the VBScript/JScript examples you'll need a browser capable of running those languages. Almost any modern browser can run JavaScript.)

Python

Python version 3 is the latest release at the time of writing. The Python download is quite big (about 13Mb for the Windows binary version) but it does include all the documentation and lots of tools, some of which we'll look at later in the tutorial. Make sure you pick the one that matches your system.

For Linux/Unix you can get the source and build it - see your sys admin!! It also comes pre-built (and pre-installed) in most Linux distributions these days and packaged versions (for Red Hat, Mandrake, Suse and Debian) can be found too. In fact you may well find that many of the system admin tools you use on Linux are actually written in Python.

The master download site for Python is:

http://www.python.org/download

Windows and MacOS users might prefer the ActiveState version which normally comes with some platform specific extras bundled with the core program. However at the time of writing the extras have not been made available for Python v3 so I recommend sticking to the official web site for now. But, by the time you read this, things might have changed, so it might be worth just checking first.

VBScript and JavaScript

As I said earlier most browsers can run JavaScript without any problems. VBScript will only work in Microsoft's Internet Explorer. You don't need to install anything for these languages, either you have them (on Windows boxes) or you don't (JavaScript only on Linux/MacOS). The only thing to watch out for is that some paranoid system administrators occasionally turn off the scripting feature of the browser for security purposes, but since so many web sites use JavaScript nowadays that's pretty unlikely.

And that's it. Bring your brain, a sense of humor and start programming....

Points to remember
• You need logical thinking and curiosity to program • Python, JavaScript and VBScript(on Windows only) are all freely available

Previous Next Contents

If you have any questions or feedback on this page send me mail at: alan.gauld@btinternet.com

What is Programming?

What will we cover?
An introduction to the terminology of computing plus some history and a brief look at the structure of a computer program.

Back to Basics

Computer Programming is the art of making a computer do what you want it to do.

At the very simplest level it consists of issuing a sequence of commands to a computer to achieve an objective. In the Microsoft world MS DOS users used to create text files with lists of commands called batch files. These simply executed the sequence of commands as a group or batch, hence the name. The files had an extension of .BAT and so were sometimes called BAT files. You can still produce these in Windows environments today but in practice they are rarely seen.

AS an example, you might be producing a document (such as this tutorial) which comprises lots of separate files. Your word processor may produce backup copies of each file as it saves a new version. At the end of the day you may want to put the current version of the document (all the latest files) into a 'backup' directory/folder. Finally, to tidy up, delete all the word processor's backup files ready to start work the next day. A simple BAT file to do this would be:

```
COPY *.HTM BACKUP
DEL *.BAK
```

If the file were called SAVE.BAT then at the end of each day I could simply type SAVE at a DOS prompt and the files would be saved and backups deleted. This is a program.

Note: Users of Linux or other operating systems have their own versions of these files often known as *shell scripts*. Unix shell scripts are much more powerful than DOS BAT files, and support most of the programming techniques that we will be discussing in this course.

Let me say that again

If you were a little daunted by that, please don't be. A computer program is simply a set of instructions to tell a computer how to perform a particular task. It's rather like a recipe: a set of instructions to tell a cook how to make a particular dish. It describes the ingredients (the data) and the sequence of steps (the process) needed to convert the ingredients into a cake or whatever. Programs are very similar in concept.

A little history

Just as you speak to a friend in a language so you 'speak' to the computer in a language. The only language that the computer understands is called *binary* and there are several different dialects of it - which is why that cool MacOS program won't run on your Windows PC and vice versa. Binary is unfortunately very difficult for humans to read or write so we have to use an intermediate language and get it translated into binary for us. This is rather like watching the American and Russian presidents talking at a summit meeting - One speaks in English, then an interpreter repeats what has been said in Russian. The other replies in Russian and the interpreter again repeats the sentence, this time in English.

Oddly enough the thing that translates our intermediate language into binary is also called an interpreter. And just as you usually need a different interpreter to translate English into Russian than you do to translate Arabic into Russian so you need a different computer interpreter to translate Python into binary from the one that translates VBScript into binary.

The very first programmers actually had to enter the binary codes themselves, this is known as *machine code* programming and is incredibly difficult. The next stage was to create a translator that simply converted English equivalents of the binary codes into binary so that instead of having to remember that the code `001273 05 04` meant `add 5 to 4` programmers could now write `ADD 5 4`. This very simple improvement made life much simpler and these systems of codes were really the first programming languages, one for each type of computer. They were known as *assembler* languages and *Assembler programming* is still used for a few specialized programming tasks today.

Even this was very primitive and still told the computer what to do at the hardware level - move data from this memory location to that memory location, add this byte to that byte etc. (Binary data is represented as a stream of binary digits or *bits*, and for convenience these are grouped into sets of eight which are called *bytes* or occasionally *octets*. Bytes traditionally were used to represent the characters of text, one byte per letter.) Programming this way was still very difficult and took a lot of programming effort to achieve even simple tasks.

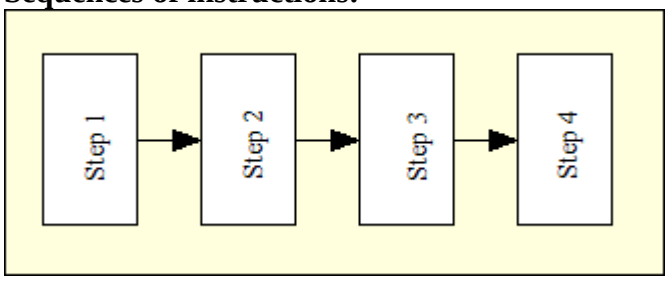
Gradually computer scientists developed higher level computer languages to make the job easier. This was just as well because at the same time users were inventing ever more complex jobs for computers to solve! This competition between the computer scientists and the users is still going on and new languages keep on appearing. This makes programming interesting but also makes it important that as a programmer you understand the concepts of programming as well as the pragmatics of doing it in one particular language.

I'll discuss some of those common concepts next, but we will keep coming back to them as we go through the course.

The common features of all programs

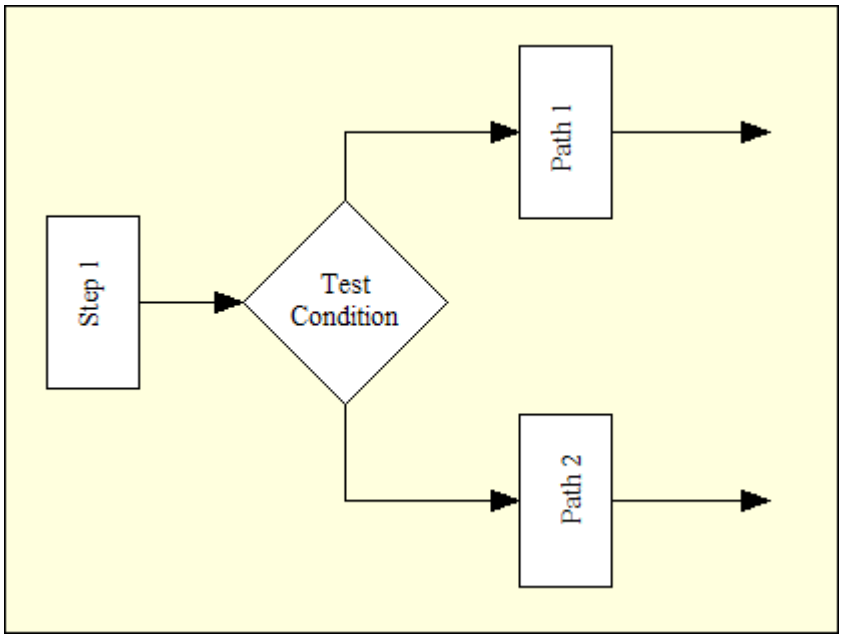
A long time ago a man called Edsger Dijkstra came up with a concept called *structured programming*. This said that all programs could be structured in the following four ways:

- **Sequences of instructions:**



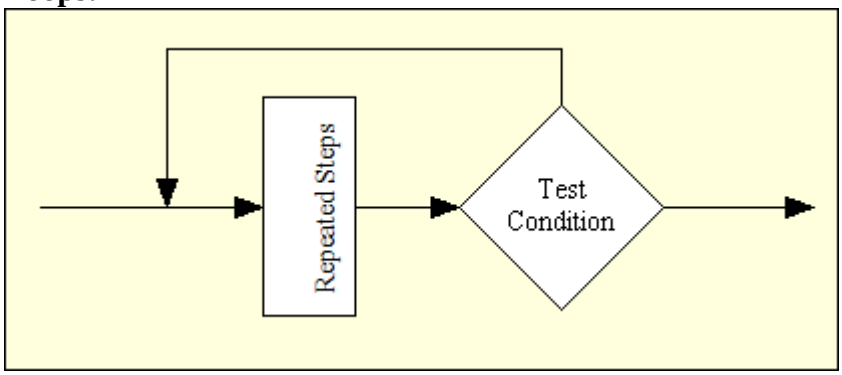
  Here the program flows from one step to the next in strict sequence.

- **Branches:**



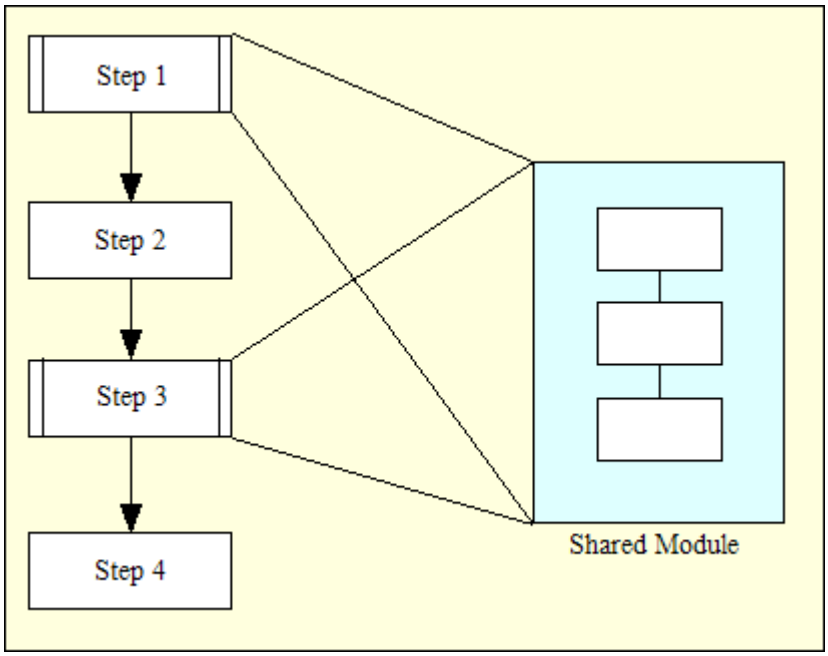
Here the program reaches a decision point and if the result of the test is true then the program performs the instructions in *Path 1*, and if false it performs the actions in *Path 2*. This is also known as a *conditional* construct because the program flow is dependent on the result of a test condition.

- **Loops:**



In this construct the program steps are repeated continuously until some test condition is reached, at which point control then flows past the loop into the next piece of program logic.

- **Modules:**



Here the program performs an identical sequence of actions several times. For convenience these common actions are placed in a module, which is a kind of mini-program which can be executed from within the main program. Other names for such a module are: *sub-routine*, *procedure* or *function*.

Along with these structures programs also need a few more features to make them useful:

- Data (we take a closer look at data in the Raw Materials topic.)
- Operations (add, subtract, compare etc.
  - we also take a look at the operations we can perform on data in the Raw Materials topic.)
- Input/Output capability (e.g. to display results
  - we look at how to read data in the "Talking to the User" and Handling Files topics.)

Once you understand those concepts and how a particular programming language implements them then you can write a program in that language.

Let's clear up some terminology

We already said that programming was the art of making a computer do what you want, but what is *a program*?

In fact there are two distinct concepts of a program. The first is the one perceived by the user - an executable file that is installed and can be run repeatedly to perform a task. For example users speak of running their "word processor program". The other concept is the program as seen by the programmer, this is the text file of instructions to the computer, written in some programming language, that can be translated into an executable file. So when you talk about a program always be clear about which concept you mean.

Basically a programmer writes a program in a high level language which is interpreted into the bytes that the computer understands. In technical speak the programmer generates *source code* and the interpreter generates *object code*. Sometimes object code has other names like: *P-Code, binary code* or *machine code*.

The translator has a couple of names, one being the *interpreter* and the other being the *compiler*. These terms actually refer to two different techniques of generating object code from source code. It used to be the case that compilers produced object code that could be run on its own (an *executable file* - another term) whereas an interpreter had to be present to run its program as it went along. The difference between these terms is now blurring however since some compilers now require interpreters to be present to do a final conversion and some interpreters simply compile their source code into temporary object code and then execute it.

From our perspective it makes no real difference, we write source code and use a tool to allow the computer to read, translate and execute it.

The structure of a program

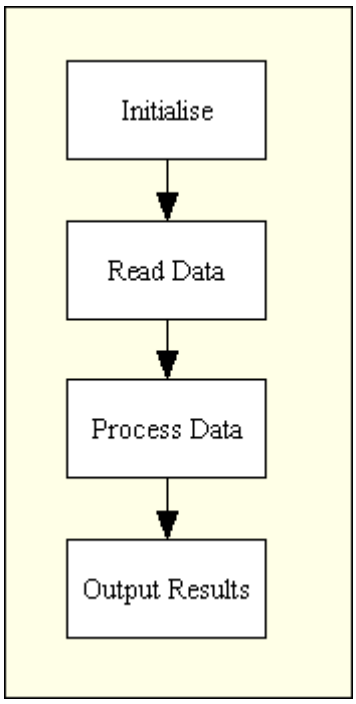
The exact structure of a program depends on the programming language and the environment that you run it on. However there are some general principles:

- A loader - every program needs to be loaded into memory by the operating system. The loader does this and is usually created by the interpreter for you.
- Data definitions - most programs operate on data and somewhere in the source code we need to define exactly what type of data we will be working with. Different languages do this very differently.
- Statements - these are the core of your program. The statements actually manipulate the data we define and do the calculations, print the output etc.

Most programs follow one of two structures:

Batch programs

These are typically started from a command line (or automatically via a scheduler utility) and tend to follow a pattern of:



That is, the program will typically start off by setting its internal state, perhaps setting totals to zero, opening the needed files etc. Once it is ready to start work it will read data either from the user by displaying prompts on a screen or from a data file. Most commonly a combination is used whereby the

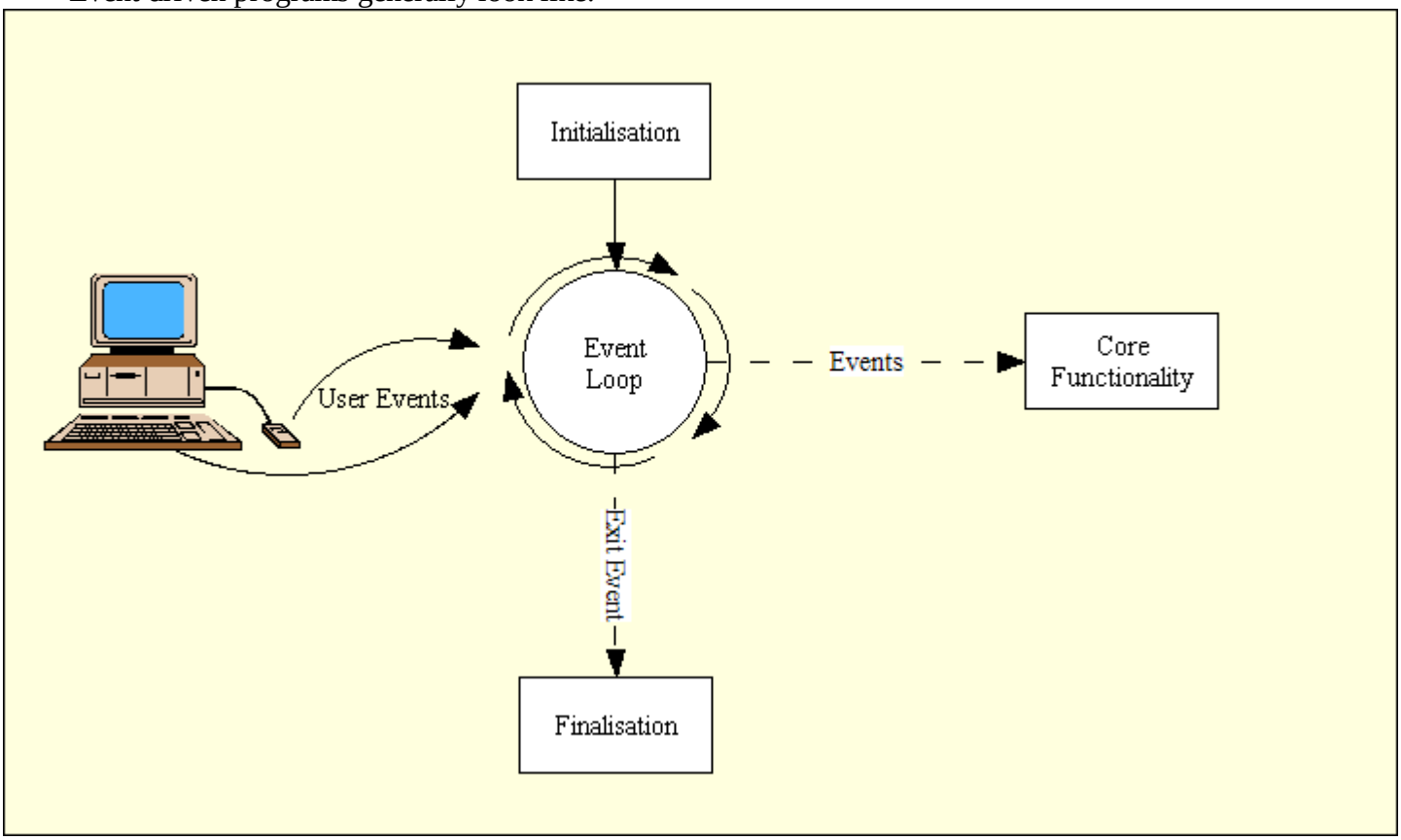
user provides the name of the data file and the real data is read from the file. Then the program does the actual data processing involving math or data conversion or whatever. Finally the results are produced, either to a screen display or, perhaps, by writing them back to a file.

All the programs we write in the early parts of this tutorial will be batch style programs.

Event driven programs

Most systems with a Graphical User Interface or GUI (eg. Microsoft Windows or MacOS) and embedded control systems - like your Microwave, camera etc. are event driven. That is the operating system sends events to the program and the program responds to these as they arrive. Events can include things a user does - like clicking the mouse or pressing a key - or things that the system itself does like updating the clock or refreshing the screen.

Event driven programs generally look like:



In this configuration the program again starts off by setting up its internal state, but then control is handed off to the event loop - which is usually provided by the operating environment (sometimes referred to as *the runtime*). The program then waits for the event loop to detect user actions which it translates to events. These events are sent to the program to deal with one at a time. Eventually the user will perform an action that terminates the program, at which point an Exit Event will be created and sent to the program.

We look at event loops and event driven programming in the "Advanced Topics" section and again in the GUI programming topic.

Points to remember

- Programs control the computer
- Programming languages allow us to 'speak' to the computer at a level that is closer to how humans think than how computers 'think'
- Programs *operate* on *data*
- Programs can be either *Batch oriented* or *Event driven*

Previous Next Contents

If you have any questions or feedback on this page send me mail at: alan.gauld@btinternet.com

Welcome to the Learning to Program Website

This web site is designed to help total beginners learn to program. There are two tutorials to choose from. "Version 2" on uses Python Version 2 as the programming language, along with VBScript and Javascript, while "Version 3" uses Python Version 3! At the time of writing I recommend that total beginners use Version 2 because Python Version 3 is still a little bit too immature, however that situation is changing rapidly and if you feel adventurous by all means give V3 a go.

Version 2 Version 3

insert Stop Press Here

If you have any questions or feedback on this page send me mail at: alan.gauld@btinternet.com

Simple Sequences

What will we cover?
• Single statements • The use of Python as a calculator • Using parentheses to get the correct result • Using format strings to print complex output • How to quit Python from within a program.

A simple sequence of instructions is the most basic program you can write. The simplest sequence is one containing a single programme statement. A statement is usually entered all on one line, although occasionally they can spill over onto two or more lines. A statement is a group of words and symbols that is meaningful to the interpreter, it's a bit like a sentence in natural language. We will try out some of these now. The examples will show what you should type at the '>>> ' Python prompt, along with the result, and the following paragraph will explain what happens.

Displaying output

The first thing we need to learn is how to get Python to display some information. Without that we would not know what the computer had done and it would all be pretty pointless!

```
>>> print('Hello there!')
Hello there!
```

Note: The first thing to notice is that you don't need to type the space between the >>> and the 'print' - Python puts that there for you. The part in bold is what you need to type, the second line is the output printed by the interpreter.

Secondly Python cares about details like whether or not you use upper or lower case. If you typed `Print` instead of `print` you would get an error because Python considers the two words to be different. (JavaScript is also fussy about case whereas VBScript is much more forgiving, but it's best to just get used to being very careful about case when programming.)

The `print()` *function* is the way to get Python to display its results to you. In this case it is printing the sequence of characters `H,e,l,l,o, ,t,h,e,r,e,!`. Such a sequence of characters is known in programming circles as a *string of characters* or a *character string* or just a plain *string*. The characters must be inside parentheses, we'll discuss the significance of those later.

You signify a string by surrounding it in quotes. In Python you can use either single quotes (as above) or double quotes: "a string". This allows you to include one type of quote within a string which is surrounded by the other type - useful for apostrophes:

```
>>> print("Monty Python's Flying Circus has a ' within it...")
Monty Python's Flying Circus has a ' within it...
```

JavaScript and VBScript are both a bit more fussy about which types of quotes you can use and where. In both of those languages I recommend sticking to double quotes where possible.

It's not just characters that can be printed:

Displaying Arithmetic Results

```
>>> print(6 + 5)
11
```

Here we have printed the result of an arithmetic operation - we added six and five. Python recognized the numbers as such and the plus sign and did the sum for us. It then printed the result.

So straight away you have a use for Python: it's a handy 'pocket calculator'!

Try a few more calculations. Use some other arithmetic *operators*:

- subtract (-)
- multiply (*)
- divide (/)

We can combine multiple expressions like this:

```
>>> print( ((6 * 5) + (7 - 5)) / (7 + 1) )
4.0
```

Notice the way I used parentheses to group the numbers together. Python sees this as:

```
((6 * 5) + (7 - 5)) / (7 + 1)
=> (30 + 2) / 8
=> 32 / 8
=> 4
```

What happens if you type the same sequence without the parentheses?

```
>>> print(6 * 5 + 7 - 5 / 7 + 1)
37.2857142857
```

This is because Python will evaluate the multiplication and division before the addition and subtraction. So Python sees it as:

```
(6*5) + 7 - (5/7)  + 1
=> 30 + 7 - 0.7143 + 1
=> 37 - 0.7143 + 1
=> 38 - 0.7143
=> 37.2857...
```

This is usually what you would expect mathematically speaking but it may not be what you expect as a programmer! Most programming languages have rules to determine the sequence of evaluation of operations and this is known as *operator precedence*. You will need to look at the reference documentation for each language to see how it works. With Python it's usually what logic and intuition would suggest, but occasionally it won't be...

As a general rule it's safest to include the parentheses to make sure you get what you want when dealing with long series of sums like this.

One other thing to note:

```
>>> print(5/2)
2.5
```

Which is pretty much what you would expect. But if you want to keep with whole numbers you can find the whole result and remainder by using the `//` sign like a division operator. Python will print the dividend:

```
>>> print(5//2)
2
```

And to get the remainder we use the modulo operator (%):

```
>>> print(5%2)
1
>>> print(7//4)
1
>>> print(7%4)
3
```

% is known as the *modulo* or *mod* operator and in other languages is often seen as `MOD` or similar. In fact in Python we can get both result(the dividend) and remainder(modulo) by using the `divmod()` function:

```
>>> print( divmod(7,4) )
(1, 3)
```

Experiment and you will soon get the idea. Why bother? Well, it turns out that so called *integer arithmetic* is very useful in programming. As a simple example we can tell whether a number is odd or even by dividing by two and checking whether the remainder was zero (i.e. it is even) or not (so it must be odd). Like this:

```
>>> print( 47 % 2 )
1
```

So we know 47 is odd. Now you could probably tell that just by looking at the last digit, 7. But imagine you were reading the data from a file or a user was typing it in. Then your program has to figure out whether it's odd or even by itself. You, the programmer, can't help it out by checking visually. That's one occasion when modulo (%) comes in very handy.

Mixing Strings and Numbers

```
>>> print( 'The total is: ', 23+45)
The total is:  68
```

You've seen that we can print strings and numbers. Now we combine the two in one print statement, separating them with a comma. We can extend this feature by combining it with a useful Python trick for outputting data called a *format string*:

```
>>> print( "The sum of %d and %d is: %d" % (7,18,7+18))
The sum of 7 and 18 is: 25
```

In this statement the format string contains '%' markers within it. These have nothing to do with the modulo operator we discussed above, instead they have a special meaning when used within a string like this. Unfortunately this double usage of `%` means you have to read carefully to determine the context and therefore what role the `%` is playing!

The letter `d` after the `%` tells Python that a 'decimal number' should be placed there. The values to fill in the markers are obtained from the values inside the parenthesised expression following the `%` sign on its own. It is important that the number of values in the final parentheses matches the number of % markers inside the string. (If this all sounds a little confusing practice a few variations on the line above and with the information following and it will soon start to make sense.)

There are other letters that can be placed after the % markers. Some of these include:

- %s - for string
- %x - for hexadecimal number
- %0.2f - for a real number with a maximum of 2 decimal places
- %04d - pad the number out to 4 digits with 0's

The Python documentation gives lots more... Note however that this style of formatting has been replaced in Python v3 by an even more powerful (but more complex) style which we will discuss in more detail when we get to the Handling Text topic later.

In fact you can print any Python *object* with the print function. Sometimes the result will not be what you hoped for (perhaps just a description of what kind of object it is) but you can always print it.

Powering Up

```
>>> import sys
```

Now this is a strange one. If you've tried it you'll see that it apparently does nothing. But that's not really true. To understand what happened we need to look at the architecture of Python (for non Python programmers, bear with me there will be a similar mechanism available to you too!)

When you start Python there are a bunch of functions and commands available to you called *built-ins*, because they are built in to the Python core. However Python can extend the list of functions available by incorporating extension modules. - It's a bit like buying a new tool in your favourite DIY store and adding it to your tool box. The tool is the `sys` part and the `import` operation puts it into the tool box.

In fact what this command does is makes available a whole bunch of new 'tools' in the shape of Python functions which are defined in a module called 'sys'. This is how Python is extended to do all sorts of clever things that are not built in to the basic system. In fact there are over a hundred modules in the *standard library* that you get with Python. You can even create your own *modules* and import and use them, just like the modules provided with Python when you installed it. We'll get to that later. There are also many more that you can download from the internet. In fact, any time you start a new project that is not covered by modules in the standard library, remember to do a Google search, there is probably something out there that can help you.

So how do we use these new tools?

A Quick exit

```
>>> sys.exit()
```

Whoops! What happened there? Simply that we executed the `exit` function defined in the `sys` module. That statement causes Python to exit.

Note 1: Normally you exit Python by typing the End Of File(*EOF*) character at the `>>>` prompt - CTRL-Z on DOS or CTRL-D on Unix. In a development tool you exit using the `File->Exit` menu etc as usual.

Note 2: If you try this inside a development tool like IDLE the tool will probably catch the attempt to exit and display a message saying something about SystemExit. Don't worry, that means the program is working and the tool is just saving you the bother of restarting from scratch.

Notice that `exit` had parentheses after it. That's because exit is a *function* defined in `sys` and when we call a Python function we need to supply the parentheses even if there's nothing inside them!

Try typing `sys.exit` without the parentheses. Python responds by telling you that exit is a function rather than by executing it!

One final thing to notice is that the last two statements are actually only useful in combination. That is, to exit from Python other than by typing EOF you need to type:

```
>>> import sys
>>> sys.exit()
```

This is a sequence of two statements! Now we're getting closer to real programming...

Using JavaScript

Unfortunately in JavaScript there is no easy way to type the commands in and see them being executed immediately as we have been doing with Python. However we can type all of the simple commands we used above into a single HTML file and load it into a browser. That way we will see what they look like in JavaScript:

```
<html><body>
<script type="text/javascript">
document.write('Hello there!<BR>');
document.write("Monty Python\'s Flying Circus has a \' within it<BR>");
document.write(6+5);
document.write("<BR>");
document.write(  ((8 * 4) + (7 - 3)) / (2 + 4)  );
document.write("<BR>");
document.write( 5/2 );
document.write("<BR>");
document.write( 5 % 2 );
</script>
</body></html>
```

And the output should look like this:

Notice that we had to write <BR> to force a new line. That's because JavaScript writes its output as HTML and HTML wraps lines into as wide a line as your browser window will allow. To force a line break we have to use the HTML symbol for a new line which is <BR>.

And VBScript too...

Like JavaScript we have to create a file with our VBScript commands and open it in a browser. The commands that we have seen, written in VBScript look like this:

```
<html><body>
<script type="text/vbscript">
MsgBox "Hello There!"
MsgBox "Monty Python's Flying Circus has a ' in it"
MsgBox 6 + 5
MsgBox ((8 * 4) + (7 - 3)) / (2 + 4)
MsgBox 5/2
MsgBox 5 MOD 2
</script>
</body></html>
```

And the output should consist of lots of dialog boxes each presenting the output from one line of the program.

One point to note is that you cannot start a string using a single quote in VBScript (We'll see why in a later topic) although you can include single quotes inside double quoted strings. To include a double quote inside a double quoted string we have to use a function called Chr which returns the character for a given ASCII character code. It's all very messy but an example should show how it works:

```
<script type="text/vbscript">
Dim qt
qt = Chr(34)
MsgBox qt & "Go Away!" & qt & " he cried"
</script>
```

Note that you can find out the ASCII code for any character by using the Character Map applet in Windows, or by visiting this web site and looking up the decimal value or, as a last resort, by using the following bit of JavaScript(!) and replacing the double quote character with the character you want:

```
<script type="text/javascript">
var code, chr = '"'; // put the character of interest here
code = chr.charCodeAt(0);
document.write("<BR>The ASCII code of  " + chr + "  is " + code);
</script>
```

Don't worry about what it means just yet, we'll get to it eventually for now just use it should you be forced to find out an ASCII value.

That's our first look at programming, it wasn't too painful was it? Before we continue though we need to take a look at the raw materials of programming, namely data and what we can do with it.

Points to remember

- Even a single command is a program
- Python does math *almost* the way you'd expect
- To get a fractional result you must use a fractional number
- You can combine text and numbers using the `%` format operator
- Quit with `import sys; sys.exit()`

Previous Next Contents

If you have any questions or feedback on this page send me mail at: alan.gauld@btinternet.com

The Raw Materials

What will we cover?
• What Data is • What Variables are • Data Types and what to do with them • Defining our own data types

Introduction

In any creative activity we need three basic ingredients: tools, materials and techniques. For example when I paint the tools are my brushes, pencils and palettes. The techniques are things like ‘washes’, wet on wet, blending, spraying etc. Finally the materials are the paints, paper and water. Similarly when I program, my tools are the programming languages, operating systems and hardware. The techniques are the programming constructs that we discussed in the previous section and the material is the data that I manipulate. In this chapter we look at the materials of programming.

This is quite a long section and by its nature you might find it a bit dry, the good news is that you don’t need to read it all at once. The chapter starts off by looking at the most basic data types available, then moves on to how we handle collections of items and finally looks at some more advanced material. It should be possible to drop out of the chapter after the collections material, cover a couple of the following chapters and then come back to this one as we start to use the more advanced bits.

Data

Data is one of those terms that everyone uses but few really understand. My dictionary defines it as:

> *"facts or figures from which conclusions can be inferred; information"*

That's not too much help but at least gives a starting point. Let’s see if we can clarify things by looking at how data is used in programming terms. Data is the “stuff”, the raw information, that your program manipulates. Without data a program cannot perform any useful function. Programs manipulate data in many ways, often depending on the *type* of the data. Each data type also has a number of *operations* - things that you can do to it. For example we’ve seen that we can add numbers together. Addition is an operation on the number type of data. Data comes in many types and we’ll look at each of the most common types and the operations available for that type:

Variables

Data is stored in the memory of your computer. You can liken this to the big wall full of boxes used in mail rooms to sort the mail. You can put a letter in any box but unless the boxes are labeled with the destination address it’s pretty meaningless. Variables are the labels on the boxes in your computer's memory.

Knowing what data looks like is fine, so far as it goes but to manipulate it we need to be able to access it and that’s what variables are used for. In programming terms we can create *instances* of data types and assign them to variables. A variable is a *reference* to a specific area somewhere in the computers memory. These areas hold the data. In some computer languages a variable must match the

type of data that it points to. Any attempt to assign the wrong type of data to such a variable will cause an error. Some programmers prefer this type of system, known as *static typing* because it can help to prevent some subtle bugs which are hard to detect.

Variable names follow certain rules dependent on the programming language. Every language has its own rules about which characters are allowed or not allowed. Some languages, including Python and JavaScript, take notice of the *case* and are therefore called *case sensitive* languages, others, like VBScript don't care. Case sensitive languages require a little bit more care from the programmer to avoid mistakes, but a consistent approach to naming variables will help a lot. One common style which we will use a lot is to start variable names with a lower case letter and use a capital letter for each first letter of subsequent words in the name, like this:

```
aVeryLongVariableNameWithCapitalisedStyle
```

We won't discuss the specific rules about which characters are legal in our languages but if you consistently use a style like that shown you shouldn't have too many problems.

In Python a variable takes the type of the data assigned to it. It will keep that type and you will be warned if you try to mix data in strange ways - like trying to add a string to a number. (Recall the example error message? It was an example of just that kind of error.) We can change the type of data that a variable points to by reassigning the variable.

```
>>> q = 7          # q is now a number
>>> print( q )
7
>>> q = "Seven"    # reassign q to a string
>>> print( q )
Seven
```

Note that the variable `q` was set to point to the number `7` initially. It maintained that value until we made it point at the character string `"Seven"`. Thus, Python variables maintain the type of whatever they point to, but we can change what they point to simply by reassigning the variable. We can check the type of a variable by using the `type()` function:

```
>>> print( type(q) )
```

At the point of reassignment the original data is 'lost' and Python will erase it from memory (unless another variable points at it too) this erasing is known as *garbage collection.*

Garbage collection can be likened to the mail room clerk who comes round once in a while and removes any packets that are in boxes with no labels. If he can't find an owner or address on the packets he throws them in the garbage. Let’s take a look at some examples of data types and see how all of this fits together.

VBScript and JavaScript variables

Both JavaScript and VBScript introduce a subtle variation in the way we use variables. In both languages it is considered good practice that variables be *declared* before being used. This is a common feature of compiled languages and of *strictly typed* languages. There is a big advantage in

doing this in that if a spelling error is made when using a variable the translator can detect that an unknown variable has been used and flag an error. The disadvantage is, of course, some extra typing required by the programmer.

VBScript

In VBScript the declaration of a variable is done via the `Dim` statement, which is short for Dimension. This is a throwback to VBScript's early roots in BASIC and in turn to Assembler languages before that. In those languages you had to tell the assembler how much memory a variable would use - its dimensions. The abbreviation has carried through from there.

A variable declaration in VBScript looks like this:

```
Dim aVariable
```

Once declared we can proceed to assign values to it just like we did in Python. We can declare several variables in the one `Dim` statement by listing them separated by commas:

```
Dim aVariable, another, aThird
```

Assignment then looks like this:

```
aVariable = 42
another = "This is a nice short sentence."
aThird = 3.14159
```

There is another keyword, `Let` that you may occasionally see. This is another throwback to BASIC and because it's not really needed you very rarely see it. In case you do, it's used like this:

```
Let aVariable = 22
```

I will not be using `Let` in this tutor.

JavaScript

In JavaScript you can pre-declare variables with the `var` keyword and, like VBScript, you can list several variables in a single `var` statement:

```
var aVariable, another, aThird;
```

JavaScript also allows you to initialize (or *define*) the variables as part of the `var` statement. Like this:

```
var aVariable = 42;
var another = "A short phrase", aThird = 3.14159;
```

This saves a little typing but otherwise is no different to VBScript's two step approach to variables. You can also declare and initialise JavaScript variables without using `var`, in the same way that you do in Python:

```
aVariable = 42;
```

But JavaScript afficianados consider it good practice to use the `var` statement, so I will do so in this tutor.

Hopefully this brief look at VBScript and JavaScript variables has demonstrated the difference between *declaration* and definition of variables. Python variables are *declared* by defining them.

Primitive Data Types

Primitive data types are so called because they are the most basic types of data we can manipulate. More complex data types are really combinations of the primitive types. These are the building blocks upon which all the other types are built, the very foundation of computing. They include letters, numbers and something called a boolean type.

Character Strings

We've already seen these. They are literally any string or sequence of characters that can be printed on your screen. (In fact there can even be non-printable *control characters* too).

In Python, strings can be represented in several ways:

With single quotes:

```
'Here is a string'
```

With double quotes:

```
"Here is a very similar string"
```

With triple double quotes:

```
""" Here is a very long string that can
    if we wish span several lines and Python will
    preserve the lines as we type them..."""
```

One special use of the latter form is to build in documentation for Python functions that we create ourselves - we'll see this later. (You can use triple single quotes but I do not recommend that since it can become hard to tell whether it is triple single quotes or a double quote and a single quote together.)

You can access the individual characters in a string by treating it as an array of characters (see arrays below). There are also usually some operations provided by the programming language to help you manipulate strings - find a sub string, join two strings, copy one to another etc.

It is worth pointing out that some languages have a separate type for characters themselves, that is for a single character. In this case strings are literally just collections of these character values. Python by contrast just uses a string of length 1 to store an individual character, no special syntax is required.

String Operators

There are a number of operations that can be performed on strings. Some of these are built in to Python but many others are provided by modules that you must import (as we did with `sys` in the Simple Sequences section).

String operators

Operator	Description
S1 + S2	Concatenation of S1 and S2
S1 * N	N repetitions of S1

We can see these in action in the following examples:

```
>>> print( 'Again and ' + 'again' )    # string concatenation
Again and again
>>> print( 'Repeat ' * 3 )                  # string repetition
Repeat Repeat Repeat
>>> print( 'Again ' + ('and again ' * 3) )  # combine '+' and '*'
Again and again and again and again
```

We can also assign character strings to variables:

```
>>> s1 = 'Again '
>>> s2 = 'and again '
>>> print( s1 + (s2 * 3) )
Again and again and again and again
```

Notice that the last two examples produced the same output.

There are lots of other things we can do with strings but we'll look at those in more detail in a later topic after we've gained a bit more basic knowledge. One important thing to note about strings in Python is that they cannot be modified. That is, you can only create a new string with some of the characters changed but you cannot directly alter any of the characters within a string. A data type that cannot be altered is known as an *immutable* type.

VBScript String Variables

In VBScript all variables are called variants, that is they can hold any type of data and VBScript tries to convert it to the appropriate type as needed. Thus you may assign a number to a variable but if you use it as a string VBScript will try to convert it for you. In practice this is similar to what Python's print command does but extended to any VBScript command. You can give VBScript a hint that you want a numeric value treated as a string by enclosing it in double quotes:

```
<script type="text/vbscript">
MyString = "42"
MsgBox MyString
</script>
```

We can join VBScript strings together, a process known as *concatenation*, using the & operator:

```
<script type="text/vbscript">
MyString = "Hello" & "World"
MsgBox MyString
</script>
```

JavaScript Strings

JavaScript strings are enclosed in either single or double quotes. In JavaScript you should *declare* variables before we use them. This is easily done using the `var` keyword. Thus to declare and *define* two string variables in JavaScript we do this:

```
<script type="text/javascript">
var aString, another;
aString = "Hello ";
another = "World";
document.write(aString + another)
</script>
```

Finally JavaScript also allows us to create String *objects*. We will discuss objects a little later in this topic but for now just think of String objects as being strings with some extra features. The main difference is that we create them slightly differently:

```
<script type="text/javascript">
var aStringObj, anotherObj;
aStringObj = String("Hello ");
anotherObj = String("World");
document.write(aStringObj + anotherObj);
</script>
```

You are probably thinking thats an awful lot of extra typing to achieve the same as before? You would be right in this case, but string objects do offer some advantages in other situations as we will see later.

Integers

Integers are whole numbers from a large negative value through to a large positive value. That's an important point to remember. Normally we don't think of numbers being restricted in size but on a computer there are upper and lower limits. The size of this upper limit is known as MAXINT and depends on the number of bits used on your computer to represent a number. On most current computers and programming languages it's 32 bits so MAXINT is around 2 billion (however VBScript is limited to about +/-32000).

Numbers with positive and negative values are known as *signed integers*. You can also get *unsigned integers* which are restricted to positive numbers, including zero. This means there is a bigger maximum number available of around 2 * MAXINT or 4 billion on a 32 bit computer since we can use the space previously used for representing negative numbers to represent more positive numbers.

Because integers are restricted in size to MAXINT adding two integers together where the total is greater than MAXINT causes the total to be wrong. On some systems/languages the wrong value is just returned as is (usually with some kind of secret flag raised that you can test if you think it might have been set). Normally an error condition is raised and either your program can handle the error or the program will exit. VBScript and JavaScript both convert the number into a different format that they can handle, albeit with a small loss of accuracy. Python is a little different in that Python uses something called a *Long Integer*, which is a Python specific feature allowing virtually unlimited size integers.

```
>>> x = 123456700 * 34567338738999
>>> print( x )
4267569568498977843300
>>> print( type(x) )
```

Notice that the result, although considered an `int` type by Python is much bigger than the value you would normally expect from a computer. The equivalent code in VBScript or JavaScript results in the number being displayed in a different format to the integer we expect. We'll find out more about that in the section on Real Numbers below.

```
<script type="text/vbscript">
Dim x
x = 123456700 * 34567338738999
MsgBox CStr(x)
</script>
```

Arithmetic Operators

We've already seen most of the arithmetic operators that you need in the 'Simple Sequences' section, however to recap:

Python Arithmetic Operators

Operator Example	Description
M + N	Addition of M and N
M - N	Subtraction of N from M
M * N	Multiplication of M and N
M / N	Division of M by N. The result will be a real number (see below)
M % N	Modulo: find the remainder of M divided by N
M**N	Exponentiation: M to the power of N

We haven't seen the last one before so let's look at an example of creating some integer variables and using the exponentiation operator:

```
>>> i1 = 2     # create an integer and assign it to i1
>>> i2 = 4
>>> i3 = i1**i2  # assign the result of 2 to the power 4 to i3
>>> print( i3 )
16
>>> print( 2**4 )  # confirm the result
16
```

Shortcut operators

One very common operation that is carried out while programming is incrementing a variable's value. Thus if we have a variable called `x` with a value of `42` and we want to increase its value to `43` we can do it like this:

```
>>> x = 42
>>> print( x )
>>> x = x + 1
>>> print( x )
```

Notice the line

```
x = x + 1
```

This is not sensible in mathematics but in programming it is. What it means is that `x` takes on the previous value of `x` plus `1`. If you have done a lot of math this might take a bit of getting used to, but basically the equal sign in this case could be read as *becomes*. So that it reads: `x` becomes `x + 1`.

Now it turns out that this type of operation is so common in practice that Python (and JavaScript) provides a shortcut operator to save some typing:

```
>>> x += 1
>>> print( x )
```

This means exactly the same as the previous assignment statement but is shorter. And for consistency similar shortcuts exist for the other arithmetic operators:

Shortcut Operators

Operator Example	Description
M += N	M = M + N
M -= N	M = M - N
M *= N	M = M * N
M /= N	M = M / N
M %= N	M = M % N

VBScript Integers

As I said earlier VBScript integers are limited to a lower value of MAXINT corresponding to a 16 bit value, namely about +/- 32000. If you need an integer bigger than that you can use a `long` integer which is the same size as a standard Python integer. There is also a `byte` type which is an 8 bit number with a maximum size of 255. In practice you will usually find the standard integer type sufficient. If the result of an operation is bigger than MAXINT then VBScript automatically converts the result to a real number (see below)

All the usual arithmetic operators are supported. Modulo is represented differently in VBScript, using the `MOD` operator. (We actually saw that in the Simple Sequences topic.) Exponentiation too is different with the caret (^) symbol being used instead of Python's `**`.

JavaScript Numbers

It will be no surprise to discover that JavaScript too has a numeric type. It too is an object as we'll describe later and its called a *Number*, original eh? :-)

A JavaScript number can also be *Not a Number* or *NaN*. This is a special version of the Number object which represents invalid numbers, usually the result of some operation which is mathematically impossible. The point of NaN is that it allows us to check for certain kinds of error without actually breaking the program. JavaScript also has special number versions to represent positive and negative infinity, a rare feature in a programming language. JavaScript number objects can be either integers or real numbers, which we look at next.

JavaScript uses mostly the same operators as Python but exponentiation is done using a special JavaScript object called `Math`. We will cover this a bit later in the tutorial when we take a closer look at modules.

Real Numbers

These include fractions. (I'm using the OED definition of fraction here. Some US correspondents tell me the US term fraction means something more specific. I simply mean any number that is not a whole number). They can represent very large numbers, much bigger than MAXINT, but with less precision. That is to say that 2 real numbers which should be identical may not seem to be when compared by the computer. This is because the computer only approximates some of the lowest details. Thus 5.0 could be represented by the computer as 4.9999999.... or 5.000000....01. These approximations are close enough for most purposes but occasionally they become important! If you get a funny result when using real numbers, bear this in mind.

Real numbers, also known as *Floating Point* numbers have the same operations as integers with the addition of the capability to truncate the number to an integer value.

Python, VBScript and JavaScript all support real numbers. In Python we create them by simply specifying a number with a decimal point in it, as we saw in the Simple Sequences topic. In VBScript and JavaScript there is no clear distinction between integers and real numbers, just use them and mostly the language will pretty much sort itself out.

Complex or Imaginary Numbers

If you have a scientific or mathematical background you may be wondering about complex numbers? If you haven't, you may not even have heard of complex numbers, in which case you can safely jump to the next heading because you don't need them! Anyhow some programming languages, including Python, provide built in support for the complex type while others provide a library of functions which can operate on complex numbers. And before you ask, the same applies to matrices too.

In Python a complex number is represented as:

```
(real+imaginaryj)
```

Thus a simple complex number addition looks like:

```
>>> M = (2+4j)
>>> N = (7+6j)
>>> print( M + N )
(9+10j)
```

All of the integer operations also apply to complex numbers.

Neither VBScript nor JavaScript offer support for complex numbers.

Boolean Values - True and False

This strange sounding type is named after a 19th century mathematician, George Boole who studied logic. Like the heading says, this type has only 2 values - either *true* or *false*. Some languages support Boolean values directly, others use a convention whereby some numeric value (often 0) represents false and another (often 1 or -1) represents true. Up until version 2.2 Python did this, however since version 2.3 Python supports Boolean values directly, using the values *True* and *False*.

Boolean values are sometimes known as "truth values" because they are used to test whether something is true or not. For example if you write a program to backup all the files in a directory you might backup each file then ask the operating system for the name of the next file. If there are no more files to save it will return an empty string. You can then test to see if the name is an empty string and store the result as a boolean value (True if it is empty, False if it isn't). You'll see how we would use that result later on in the tutorial.

Boolean (or Logical) Operators

Operator Example	Description	Effect
A and B	AND	True if A,B are both True, False otherwise.
A or B	OR	True if either or both of A,B are true. False if both A and B are false
A == B	Equality	True if A is equal to B
A != B or A <> B	Inequality	True if A is NOT equal to B.
not B	Negation	True if B is not True

Note: the last one operates on a single value, the others all compare two values.

VBScript, like Python has a Boolean type with the values True and False.

JavaScript also supports a Boolean type but this time the values are *true* and *false* (note, with a lowercase first letter).

Finally the different languages have slightly different names for the Boolean type internally, in Python it is *bool*, in VBScript and JavaScript it is *Boolean*. Most of the time you won't need to worry about that because we tend not to create variables of Boolean types but simply use the results in tests.

Collections

Computer science has built a whole discipline around studying collections and their various behaviors. Sometimes collections are called *containers* or *sequences*. In this section we will look first of all at the collections supported in Python, VBScript and JavaScript, then we'll conclude with a brief summary of some other collection types you might come across in other languages.

List

We are all familiar with lists in everyday life. A list is just a sequence of items. We can add items to a list or remove items from the list. Usually, where the list is written paper we can't insert items in the middle of a list only at the end. However if the list is in electronic format - in a word processor say - then we can insert items anywhere in the list.

We can also search a list to check whether something is already in the list or not. But you have to find the item you need by stepping through the list from front to back checking each item to see if it's the item you want. Lists are a fundamental collection type found in many modern programming languages.

Python lists are built into the language. They can do all the basic list operations we discussed above and in addition have the ability to *index* the elements inside the list. By indexing I mean that we can refer to a list element by its sequence number (assuming the first element starts at zero).

In VBScript there are no lists as such but other collection types which we discuss later can simulate their features.

In JavaScript there are no lists as such but almost everything you need to do with a list can be done using a JavaScript *array* which is another collection type that we discuss a little later.

List operations

Python provides many operations on collections. Nearly all of them apply to Lists and a subset apply to other collection types, including strings which are just a special type of list - a list of characters. To create and access a list in Python we use square brackets. You can create an empty list by using a pair of square brackets with nothing inside, or create a list with contents by separating the values with commas inside the brackets:

```
>>> aList = []
>>> another = [1,2,3]
>>> print( another )
[1, 2, 3]
```

We can access the individual elements using an index number, where the first element is 0, inside square brackets. For example to access the third element, which will be index number 2 since we start from zero, we do this:

```
>>> print( another[2] )
3
```

We can also change the values of the elements of a list in a similar fashion:

```
>>> another[2] = 7
>>> print( another )
[1, 2, 7]
```

Notice that the third element (index 2) changed from 3 to 7.

You can use negative index numbers to access members from the end of the list. This is most commonly done using -1 to get the last item:

```
>>> print( another[-1] )
7
```

We can add new elements to the end of a list using the `append()` operation:

```
>>> aList.append(42)
>>> print( aList )
[42]
```

We can even hold one list inside another, thus if we append our second list to the first:

```
>>> aList.append(another)
>>> print( aList )
[42, [1, 2, 7]]
```

Notice how the result is a list of two elements but the second element is itself a list (as shown by the []'s around it). We can now access the element 7 by using a double index:

```
>>> print( aList[1][2] )
7
```

The first index, 1, extracts the second element which is in turn a list. The second index, 2, extracts the third element of the sublist.

This nesting of lists one inside the other is extremely useful since it effectively allows us to build tables of data, like this:

```
>>> row1 = [1,2,3]
>>> row2 = ['a','b','c']
>>> table = [row1, row2]
>>> print( table )
[ [1,2,3], ['a','b','c'] ]
>>> element2 = table[0][1]
>>> print( element2 )
2
```

We could use this to create an address book where each entry was a list of name and address details. For example, here is such an address book with two entries:

```
>>> addressBook = [
... ['Fred', '9 Some St',' Anytown', '0123456789'],
... ['Rose', '11 Nother St', 'SomePlace', '0987654321']
... ]
>>>
```

Notice that we constructed the nested list all on one line. That is because Python sees that the number of opening and closing brackets don't match and keeps on reading input until they do. This can be a very effective way of quickly constructing complex data structures while making the overall structure - a list of lists in this case - clear to the reader. (If you are using IDLE you won't see the `...` prompt, just a blank line.)

As an exercise try extracting Fred's telephone number - element 3, from the first row - remembering that the indexes start at zero. Also try adding a few new entries of your own using the `append()` operation described above.

Note that when you exit Python your data will be lost, however you will find out how to preserve it once we reach the topic on files.

The opposite of adding elements is, of course, removing them and to do that we use the `del` command:

```
>>> del aList[1]
>>> print( aList )
[42]
```

Notice that `del` does not require parentheses around the value, unlike the print function. This is because `del` is technically a *command* not a *function*. The distinction is subtle and you can put parentheses around the value for consistency if you prefer, it will still work OK.

If we want to join two lists together to make one we can use the same concatenation operator '+' that we saw for strings:

```
>>> newList = aList + another
>>> print( newList )
[42, 1, 2, 7]
```

Notice that this is slightly different to when we appended the two lists earlier, then there were 2 elements, the second being a list, this time there are 4 elements because the elements of the second list have each, individually, been added to `newList`. This time if we access element 1, instead of getting a sublist, as we did previously, we will only get `1` returned:

```
>>> print( newList[1] )
1
```

We can also apply the multiplication sign as a repetition operator to populate a list with multiples of the same value:

```
>>> zeroList = [0] * 5
>>> print( zeroList )
[0, 0, 0, 0, 0]
```

We can find the index of a particular element in a list using the `index()` operation, like this:

```
>>> print( [1,3,5,7].index(5) )
2
>>> print( [1,3,5,7].index(9) )
Traceback (most recent call last):
  File "", line 1, in <module>
ValueError: list.index(x): x not in list
```

Notice that trying to find the index of something that's not in the list results in an error. We will look at ways to test whether something is in a list or not in a later topic.

Finally, we can determine the length of a list using the built-in `len()` function:

```
>>> print( len(aList) )
1
>>> print( len(newList) )
4
>>> print( len(zeroList) )
```

```
5
```

Neither JavaScript nor VBScript directly support a list type although as we will see later they do have an Array type that can do many of the things that Python's lists can do.

Tuple

Not every language provides a tuple construct but in those that do it's extremely useful. A tuple is really just an arbitrary collection of values which can be treated as a unit. In many ways a tuple is like a list, but with the significant difference that tuples are *immutable* which, you may recall, means that you can't change them nor append to them once created. In Python, tuples are simply represented by parentheses containing a comma separated list of values, like so:

```
>>> aTuple = (1,3,5)
>>> print( aTuple[1] )    # use indexing like a list
3
>> aTuple[2] = 7       # error, can't change a tuple's elements
Traceback (innermost last):
  File "", line 1, in ?
       aTuple[2] = 7
TypeError: object doesn't support item assignment
```

The main things to remember are that while parentheses are used to define the tuple, square brackets are used to index it and you can't change a tuple once it's created. Otherwise most of the list operations also apply to tuples.

Finally, although you cannot change a tuple you can effectively add members using the addition operator because this actually creates a new tuple. Like this:

```
>>> tup1 = (1,2,3)
>>> tup2 = tup1 + (4,) # comma to make it a tuple rather than integer
>>> print( tup2 )
(1,2,3,4)
```

If we didn't use the trailing comma after the 4 then Python would have interpreted it as the integer 4 inside parentheses, not as a true tuple. But since you can't add integers to tuples it results in an error, so we add the comma to tell Python to treat the parentheses as a tuple. Any time you need to persuade Python that a single entry tuple really is a tuple add a trailing comma as we did here.

Neither VBScript nor JavaScript have any concept of tuples.

Dictionary or Hash

In the same way that a literal dictionary associates a meaning with a word a dictionary type contains a value associated with a key, which may or may not be a string. The value can be retrieved by 'indexing' the dictionary with the key. Unlike a literal dictionary, the key doesn't need to be a character string (although it often is) but can be any immutable type including numbers and tuples. Similarly the values associated with the keys can have any kind of data type. Dictionaries are usually implemented internally using an advanced programming technique known as a *hash table*. For that reason a dictionary may sometimes be referred to as a hash. This has nothing to do with drugs! :-)

Because access to the dictionary values is via the key, you can only put in elements with unique keys. Dictionaries are immensely useful structures and are provided as a built-in type in Python although in many other languages you need to use a module or even build your own. We can use dictionaries in lots of ways and we'll see plenty examples later, but for now, here's how to create a dictionary in Python, fill it with some entries and read them back:

```
>>> dct = {}
>>> dct['boolean'] = "A value which is either true or false"
>>> dct['integer'] = "A whole number"
>>> print( dct['boolean'] )
A value which is either true or false
```

Notice that we initialize the dictionary with braces, then use square brackets to assign and read the values.

Just as we did with lists we can initialize a dictionary as we create it using the following format:

```
>>> addressBook = {
... 'Fred' : ['Fred', '9 Some St',' Anytown', '0123456789'],
... 'Rose' : ['Rose', '11 Nother St', 'SomePlace', '0987654321']
... }
>>>
```

The key and value are separated by a colon and the pairs are separated by commas.

You can also specify a dictionary using a slightly different format (see below), which style you prefer is mainly a matter of taste!

```
>>> book = dict(Fred=['Fred', '9 Some St',' Anytown', '0123456789'],
...             Rose=['Rose', '11 Nother St', 'SomePlace', '0987654321'])
>>> print( book['Fred'][3] )
0123456789
```

Notice you don't need quotes around the key in the definition because Python assumes it is a string (but you still need them to extract the values). In practice this limits its usefulness so I tend to prefer the first version using braces.

Either way we have made our address book out of a dictionary which is keyed by name and stores our lists as the values. Rather than work out the numerical index of the entry we want we can just use the name to retrieve all the information, like this:

```
>>> print( addressBook['Rose'] )
['Rose', '11 Nother St', 'SomePlace', '0987654321']
>>> print( addressBook['Fred'][3] )
0123456789
```

In the second case we indexed the returned list to get only the telephone number. By creating some variables and assigning the appropriate index values we can make this much easier to use:

```
>>> name = 0
>>> street = 1
>>> town = 2
```

```
>>> tel = 3
```

And now we can use those variables to find out Rose's town:

```
>>> print( addressBook['Rose'][town] )
SomePlace
```

Notice that whereas `'Rose'` was in quotes because the key is a string, the `town` is not because it is a variable name and Python will convert it to the index value we assigned, namely 2. At this point our Address Book is beginning to resemble a usable database application, thanks largely to the power of dictionaries. It won't take a lot of extra work to save and restore the data and add a query prompt to allow us to specify the data we want. We will do that as we progress through the other tutorial topics.

Of course we could use a dictionary to store the data too, then our address book would consist of a dictionary whose keys were the names and the values were dictionaries whose keys were the field names, like this:

```
addressBook = {
... 'Fred' : {'name':      'Fred',
...           'street':    '9 Some St',
...           'town':      'Anytown',
...           'tel':       '0123456789'},
... 'Rose' : {'name':      'Rose',
...           'street':    '11 Nother St',
...           'town':      'SomePlace',
...           'tel':       '0987654321'}
... }
```

Notice that this is a very readable format although it requires a lot more typing. Data stored in a format where its meaning and content are combined in a human readable format is often referred to as *self-documenting* data. Also, when we include a data structure inside another identical structure - a dictionary inside a dictionary in this case - we call that *nesting* and the inner dictionary would be called the *nested dictionary*.

In practice we access this data in a very similar way to the list with named indexes:

```
>>> print( addressBook['Rose']['town'] )
SomePlace
```

Notice the extra quotes around `town`. Otherwise it's exactly the same. One advantage of using this approach is that we can insert new fields and the existing code will not break whereas with the named indexes we would need to go back and change all of the index values. If we used the same data in several programs that could be a lot of work. Thus a little bit of extra typing now could save us a lot of extra effort in the future.

Due to their internal structure dictionaries do not support very many of the collection operators that we've seen so far. None of the concatenation, repetition or appending operations work. (Although you can of course assign new key/value pairs directly as we saw at the beginning of the section.) To assist us in accessing the dictionary keys there is an operation that we can use, `keys()`, which returns a list of all the keys in a dictionary. For example to get a list of all the names in our address book we could do:

```
>>> print( addressBook.keys() )
['Fred','Rose']
```

Note however that dictionaries do not store their keys in the order in which they are inserted so you may find the keys appear in a strange order, indeed the order may even change over time. Don't worry about that, you can still use the keys to access your data and the right value will still come out OK.

VBScript Dictionaries

VBScript provides a dictionary object which offers similar facilities to the Python dictionary but the usage is slightly different. To create a VBScript dictionary we have to declare a variable to hold the object, then create the object, finally we can add entries to the new dictionary, like this:

```
Dim dict     ' Create a variable.
Set dict = CreateObject("Scripting.Dictionary")
dict.Add "a", "Athens" ' Add some keys and items.
dict.Add "b", "Belgrade"
dict.Add "c", "Cairo"
```

Notice that the `CreateObject` function specifies that we are creating a `"Scripting.Dictionary"` object, that is a `Dictionary` object from the VBScript's `Scripting` module. Don't worry too much about that for now, we'll discuss it in more depth when we look at objects later in the tutor. Hopefully you can at least recognize and recall the concept of using an object from a module from the simple sequences topic earlier. The other point to notice is that we must use the keyword `Set` when assigning an object to a variable in VBScript.

Now we access the data like so:

```
item = dict.Item("c") ' Get the item.
dict.Item("c") = "Casablanca" ' Change the item
```

There are also operations to remove an item, get a list of all the keys, check that a key exists etc.

Here is a complete but simplified version of our address book example in VBScript:

```
<script type="text/VBScript">
Dim addressBook
Set addressBook = CreateObject("Scripting.Dictionary")
addressBook.Add "Fred", "Fred, 9 Some St, Anytown, 0123456789"
addressBook.Add "Rose", "Rose, 11 Nother St, SomePlace, 0987654321"

MsgBox addressBook.Item("Rose")
</script>
```

This time, instead of using a list, we have stored all the data as a single string. (This of course makes it much harder to extract individual fields as we did with the list or dictionary.) We then access and print Rose's details in a message box.

JavaScript Dictionaries

JavaScript doesn't really have a dictionary object of its own, although if you are using Internet Explorer you can get access to the VBScript `Scripting.Dictionary` object discussed above, with all of the same facilities. But since it's really the same object I won't cover it further here. Finally JavaScript arrays can be used very much like dictionaries but we'll discuss that in the array section below.

If you're getting a bit fed up, you can jump to the next chapter at this point. Remember to come back and finish this one when you start to come across types of data we haven't mentioned so far.

Other Collection Types

Array or Vector

The array is one of the earlier collection types in computing history. It is basically a list of items which are indexed for easy and fast retrieval. Usually you have to say up front how many items you want to store and usually you can only store data of a single type. These fixed size and fixed type features are what distinguishes arrays from the list data type discussed above. (Notice I said "usually" above. That's because different languages have widely different ideas of what exactly constitutes an array that it is hard to make definite rules.)

Python supports arrays through a module but it is rarely needed because the built in list type can usually be used instead. VBScript and JavaScript both have arrays as a data type, so let's briefly look at how they are used:

VBScript Arrays

In VBScript an array is a fixed length collection of data accessed by a numerical index. It is declared and accessed like this:

```
Dim AnArray(42)    ' A 43! element array
AnArray(0) = 27    ' index starts at 0
AnArray(1) = 49
myVariable = AnArray(1) ' read the value
```

Note the use of the `Dim` keyword. This *dimensions* the variable. This is a way of telling VBScript about the variable, if you start your script with `OPTION EXPLICIT` VBScript will expect you to `Dim` any variables you use, which many programming experts believe is good practice and leads to more reliable programs. Also notice that we specify the last valid index, `42` in our example, which means the array actually has 43 elements because it starts at 0.

Notice also that in VBScript we use parentheses to dimension and index the array, not the square brackets used in Python and, as we'll soon see, JavaScript. Finally, recall that I said arrays usually only store one type of data? Well in VBScript there is only one official type of data: the `Variant`, which in turn can store any kind of VBScript value. So a VBScript array only stores Variants, which, in practice, means they can store anything! Confusing? It is if you think about it too much, so don't, just use them!

As with Python lists, we can declare multiple dimensional arrays to model tables of data, for our address book example:

```
Dim MyTable(2,3)  ' 3 rows, 4 columns
MyTable(0,0) = "Fred"  ' Populate Fred's entry
MyTable(0,1) = "9 Some Street"
```

```
MyTable(0,2) = "Anytown"
MyTable(0,3) = "0123456789"
MyTable(1,0) = "Rose"  ' And now Rose...
...and so on...
```

Unfortunately there is no way to populate the data all in one go as we did with Python's lists, we have to populate each field one by one. If we combine VBScript's dictionary and array capability we get almost the same usability as we did with Python. It looks like this:

```
<script type="text/VBScript">
Dim addressBook
Set addressBook = CreateObject("Scripting.Dictionary")
Dim Fred(3)
Fred(0) = "Fred"
Fred(1) = "9 Some St"
Fred(2) = "Anytown"
Fred(3) = "0123456789"
addressBook.Add "Fred", Fred

MsgBox addressBook.Item("Fred")(3) ' Print the Phone Number
</script>
```

The final aspect of VBScript arrays that I want to consider is the fact that they don't need to be fixed in size at all! However this does not mean we can just arbitrarily keep adding elements as we did with our lists, rather we can explicitly resize an array. For this to happen we need to declare a *Dynamic array* which we do, quite simply by omitting the size, like this:

```
Dim DynArray()  ' no size specified
```

To resize it we use the `ReDim` command, like so:

```
<script type="text/vbscript">
Dim DynArray()
ReDim DynArray(5)  ' Initial size = 5
DynArray(0) = 42
DynArray(4) = 26
MsgBox "Before: " & DynArray(4)  ' prove that it worked
' Resize to 21 elements keeping the data we already stored
ReDim Preserve DynArray(20)
DynArray(15) = 73
MsgBox "After Preserve: " & DynArray(4) & " " & DynArray(15)' Old and new still t
' Resize to 51 items but lose all data
Redim DynArray(50)
MsgBox "Without Preserve: " & DynArray(4) & " Oops, Where did it go?"
</script>
```

As you can see this is not so convenient as a list which adjusts its length automatically, but it does give the programmer more control over how the program behaves. This level of control can, amongst other things improve security since some viruses can exploit dynamically re-sizable data stores.

JavaScript Arrays

Arrays in JavaScript are in many ways a misnomer. They are called arrays but are actually a curious mix of the features of lists, dictionaries and traditional arrays. At the simplest level we can declare a new Array of 10 items of some type, like so:

```
var items = new Array(10);
```

Notice the use of the keyword `new` to create the Array. This is similar in effect to the `CreateObject()` function we used in VBScript to create a dictionary. Also notice that we use parentheses to define the size of the array.

We can now populate and access the elements of the array like this:

```
items[4] = 42;
items[7] = 21;
var aValue = items[4];
```

So once again we use square brackets to access the array elements. And once again the indexes start from zero.

However JavaScript arrays are not limited to storing a single type of value, we can assign anything to an array element:

```
items[9] = "A short string";
var msg = items[9];
```

Also we can create arrays by providing a list of items, like so:

```
var moreItems = new Array("one","two","three",4,5,6);
aValue = moreItems[3];
msg = moreItems[0];
```

Another feature of JavaScript arrays is that we can determine the length through a hidden property called `length`. We access the length like this:

```
var size = items.length;
```

Notice that once again the syntax for this uses an `name.property` format and is very like calling a function in a Python module but without the parentheses.

As mentioned, JavaScript arrays start indexing at zero by default. However, JavaScript array indexes are not limited to numbers, we can use strings too, and in this case they become almost identical to dictionaries! We can also extend an array by simply assigning a value to an index beyond the current maximum - which means we don't really need to specify a size when we create one, even though it is considered good practice! We can see these features in use in the following code segment:

```
<script type="text/javascript">
var items = new Array(10);
var moreItems = new Array(1);
items[42] = 7;
moreItems["foo"] = 42;
msg = moreItems["foo"];
document.write("msg = " + msg + " and items[42] = " + items[42] );
</script>
```

Finally, let's look at our address book example once more, this time using JavaScript arrays:

```
<script type="text/javascript">
var addressBook = new Array();
addressBook["Fred"] = new Array("Fred", "9 Some St", "Anytown", "0123456789");
addressBook["Rose"] = new Array("Rose", "11 Nother St", "SomePlace", "0987654321"

document.write(addressBook.Rose);
</script>
```

Notice that we can also access the key as if it were a property like `length`. JavaScript arrays really are quite remarkably flexible data structures!

Stack

Think of a stack of trays in a restaurant. A member of staff puts a pile of clean trays on top and these are removed one by one by customers. The trays at the bottom of the stack get used last (and least!). Data stacks work the same way: you *push* an item onto the stack or *pop* one off. The item popped is always the last one pushed. This property of stacks is sometimes called *Last In First Out* or *LIFO*. One useful property of stacks is that you can reverse a list of items by pushing the list onto the stack then popping it off again. The result will be the reverse of the starting list. Stacks are not built in to Python, VBScript or JavaScript. You have to write some program code to implement the behavior. Lists are usually the best starting point since like stacks they can grow as needed.

Try writing a stack using a Python list. Remember that you can `append()` to the end of a list and `del()` items at a given index. Also you can use `-1` to index the last item in a list. Armed with that information you should be able to write a program that pushes 4 characters onto a list and then pops them off again, printing them as you go. Just watch which order you call print and del! If you get it right then they should print in the reverse order to how you pushed them on.

Bag

A bag is a collection of items with no specified order and it can contain duplicates. Bags usually have operators to enable you to add, find and remove items. In our languages bags are just lists.

Set

A set has the property of only storing one of each item. You can usually test to see if an item is in a set (membership), add or remove items and join two sets together in various ways corresponding to set theory in math (e.g. union, intersect etc). Sets do not have any concept of order. VBScript and JavaScript do not implement sets directly but you can approximate the behavior fairly easily using dictionaries.

In Python sets are supported as a native data type.

The basic usage is like this:

```
>>> A = set()  # create an empty set
>>> B = set([1,2,3]) # a set from a list
>>> C = {3,4,5} # initialisation, like [] in lists
>>> D = {6,7,8}
>>> # Now try out some set operations
>>> print( B.union(C) )
{1, 2, 3, 4, 5}
>>> print( B.intersection(C) )
{3}
```

```
>>> print( B.issuperset({2}) )
True
>>> print( {3}.issubset(C) )
True
>>> print( C.intersection(D) == A )
True
```

There are short hand versions of union and intersection too:

```
>>> print( B & C ) #  same as B.intersection(C)
>>> print( B | C ) #  same as B.union(C)
```

And finally you can test whether an item is in a set using the 'in' operator:

```
>>> print( 2 in B )
True
```

There are a number of other set operations but these should be enough for now.

Queue

A queue is rather like a stack except that the first item into a queue is also the first item out. This is known as *First In First Out* or *FIFO* behavior. This is usually implemented using a list or array.

See if you can write a stack using a list. Remember you can add to a list with `append()` and delete from a given position using `del()`. Try to add 4 characters to your stack and then get them out and print them. They should print in the same order that you inserted them.

There's a whole bunch of other collection types but the ones we have covered are the main ones that you are likely to come across. (And in fact we'll only be using a few of the ones we've discussed in this tutor, but you will see the others mentioned in articles and in programming discussion groups!)

Files

As a computer user you should be very familiar with files - they form very basis of nearly everything we do with computers. It should be no surprise then, to discover that most programming languages provide a special *file* type of data. However files and the processing of them are so important that I will put off discussing them till later when they get a whole topic to themselves.

Dates and Times

Dates and times are often given dedicated types in programming. At other times they are simply represented as a large number (typically the number of seconds from some arbitrary date/time, such as when the operating system was written!). In other cases the data type is what is known as a complex type as described in the next section. This usually makes it easier to extract the month, day, hour etc. We will take a brief look at using the Python `time` module in a later topic. Both VBScript and JavaScript have their own mechanisms for handling time but I won't be discussing them further.

Complex/User Defined

Sometimes the basic types described above are inadequate even when combined in collections. Sometimes, what we want to do is group several bits of data together then treat it as a single item. An example might be the description of an address:

a house number, a street and a town. Finally there's the post code or zip code.

Most languages allow us to group such information together in a *record* or *structure* or with the more modern, object oriented version, a *class*.

VBScript

In VBScript such a record definition looks like:

```
Class Address
     Public HsNumber
     Public Street
     Public Town
     Public ZipCode
End Class
```

The `Public` keyword simply means that the data is accessible to the rest of the program, it's possible to have `Private` data too, but we'll discuss that later in the course.

Python

In Python it's only a little different:

```
>>> class Address:
...   def __init__(self, Hs, St, Town, Zip):
...     self.HsNumber = Hs
...     self.Street = St
...     self.Town = Town
...     self.ZipCode = Zip
...
```

That may look a little arcane but don't worry I'll explain what the `def __init__(...)` and `self` bits mean in the section on object orientation. One thing to note is that there are two underscores at each end on `__init__`. This is a Python convention that we will discuss later. Also you need to use the spacing shown above, as we'll explain later Python is a bit picky about spacing. For now just make sure you copy the layout above.

Some people have had problems trying to type this example at the Python prompt. At the end of this chapter you will find a box with more explanation, but you can just wait till we get the full story later in the course if you prefer. If you do try typing it into Python then please make sure you copy the indentation shown. As you'll see later Python is very particular about indentation levels.

The main thing I want you to recognize in all of this is that, just as we did in VBScript, we have gathered several pieces of related data into a single structure called `Address`.

JavaScript

JavaScript provides a slightly strange name for its structure format, namely `function`! Now functions are normally associated with operations not collections of data however in JavaScript's case it can cover either. To create our address object in JavaScript we do this:

```
function Address(Hs,St,Town,Zip)
{
   this.HsNum = Hs;
```

```
   this.Street = St;
   this.Town = Town;
   this.ZipCode = Zip;
}
```

Once again, ignore the syntax and use of the keyword `this`, the end result is a group of data items that we call Address and can treat as a single unit.

OK, So we can create these data structures but what can we do with them once created? How do we access the data items inside? That's our next mission.

Accessing Complex Types

We can assign a complex data type to a variable too, but to access the individual *fields* of the type we must use some special access mechanism (which will be defined by the language). Usually this is a dot.

Using VBScript

To consider the case of the address class we defined above we would do this in VBScript:

```
Dim Addr
Set Addr = New Address

Addr.HsNumber = 7
Addr.Street = "High St"
Addr.Town = "Anytown"
Addr.ZipCode = "123 456"

MsgBox Addr.HsNumber & " " & Addr.Street & " " & Addr.Town
```

Here we first of all *Dimension* a new variable, `Addr`, using `Dim` then we use the `Set` keyword to create a new *instance* of the `Address` class. Next we assign values to the fields of the new address instance and finally we print out the address in a Message Box.

And in Python

And in Python, assuming you have already typed in the class definition above:

```
>>> Addr = Address(7,"High St","Anytown","123 456")
>>> print( Addr.HsNumber, Addr.Street, Addr.Town )
7 High St Anytown
```

Which creates an instance of our `Address` type and assigns it to the variable `Addr`. In Python we can pass the field values to the new *object* when we create it. We then print out the `HsNumber` and `Street` fields of the newly created instance using the dot operator. You could, of course, create several new Address instances each with their own individual values of house number, street etc. Why not experiment with this yourself? Can you think of how this could be used in our address book example from earlier in the topic?

JavaScript too

The JavaScript mechanism is very similar to the others but has a couple of twists, as we'll see in a moment. However the basic mechanism is straightforward and the one I recommend you use:

```
var addr = new Address(7, "High St", "Anytown", "123 456");
document.write(addr.HsNum + " " + addr.Street + " " + addr.Town);
```

One final mechanism that we can use in JavaScript is to treat the object like a dictionary and use the field name as a key:

```
document.write( addr['HsNum'] + " " + addr['Street'] + " " +  addr['Town']);
```

I can't really think of any good reason to use this form other than if you were to be given the field name as a string, perhaps after reading a file or input from the user of your program (we'll see how to do that later too).

User Defined Operators

User defined types can, in some languages, have operations defined too. This is the basis of what is known as *object oriented programming*. We dedicate a whole section to this topic later but essentially an object is a collection of data elements and the operations associated with that data, wrapped up as a single unit. Python uses objects extensively in its standard library of modules and also allows us as programmers to create our own object types.

Object operations are accessed in the same way as data members of a user defined type, via the dot operator, but otherwise look like functions. These special functions are called *methods*. We have already seen this with the `append()` operation of a list. Recall that to use it we must tag the function call onto the variable name:

```
>>> listObject = []    # an empty list
>>> listObject.append(42) # a method call of the list object
>>> print( listObject )
[42]
```

When an object type, known as a class, is provided in a Python module we must import the module (as we did with `sys` earlier), then prefix the object type with the module name when creating an instance that we can store in a variable (while still using the parentheses, of course). We can then use the variable without using the module name.

We will illustrate this by considering a fictitious module meat which provides a Spam class. We import the module, create an instance of Spam, assigning it the name `mySpam` and then use `mySpam` to access its operations and data like so:

```
>>> import meat
>>> mySpam = meat.Spam()  # create an instance, use module name
>>> mySpam.slice()        # use a Spam operation
>>> print( mySpam.ingredients )  # access Spam data
{"Pork":"40%", "Ham":"45%", "Fat":"15%"}
```

In the first line we import the (non-existent!) module `meat` into the program. In the second line we use the meat module to create an instance of the Spam class - by calling it as if it were a function! In the third line we access one of the Spam class's operations, `slice()`, treating the object (`mySpam`) as if it were a module and the operation were in the module. Finally we access some data from within the `mySpam` object using the same module like syntax. We will be looking at real examples of this (i.e. ones that work!) later in the course.

Other than the need to create an instance, there's no real difference between using objects provided within modules and functions found within modules. Think of the object name simply as a label which keeps related functions and variables grouped together.

Another way to look at it is that objects represent real world things, to which we as programmers can do things. That view is where the original idea of objects in programs came from: writing computer simulations of real world situations.

Both VBScript and JavaScript work with objects and in fact that's exactly what we have been using in each of the Address examples above. We have defined a class and then created an instance which we assigned to a variable so that we could access the instance's properties. Go back and review the previous sections in terms of what we've just said about classes and objects. Think about how classes provide a mechanism for creating new types of data in our programs by binding together the data and operations of the new type.

Python Specific Operators

In this tutor my primary objective is to teach you to program and, although I use Python in the tutor, there is no reason why, having read this, you couldn't go out and read about another language and use that instead. Indeed that's exactly what I expect you to do since no single programming language, even Python, can do everything. However, because of that objective, I do not teach all of the features of Python but focus on those which can generally be found in other languages too. As a result there are several Python specific features which, while they are quite powerful, I don't describe at all, and that includes special operators. Most programming languages have operations which they support and other languages do not. It is often these 'unique' operators that bring new programming languages into being, and certainly are important factors in determining how popular the language becomes.

For example Python supports such relatively uncommon operations as list slicing (`spam[X:Y]`) for extracting a section (or slice) out from the middle of a list(or string, or tuple) and tuple assignment (`X, Y = 12, 34`) which allows us to assign multiple variable values at one time.

It also has the facility to perform an operation on every member of a collection using its `map()` function which we describe in the Functional Programming topic. There are many more and it's often said that "Python comes with the batteries included". For details of how most of these Python specific operations work you'll need to consult the Python documentation.

Finally, it's worth pointing out that although I say they are Python specific, that is not to say that they can't be found in any other languages but rather that they will not *all* be found in every language. The operators that we cover in the main text are generally available in some form in virtually all modern programming languages.

That concludes our look at the raw materials of programming, let's move onto the more exciting topic of technique and see how we can put these materials to work.

More information on the Address example

Although, as I said earlier, the details of this example are explained later, some readers have found difficulty getting the Python example to work. This note gives a line by line explanation of the Python code. The complete code for the example looks like this:

```
>>> class Address:
```

```
...   def __init__(self, Hs, St, Town, Zip):
...     self.HsNumber = Hs
...     self.Street = St
...     self.Town = Town
...     self.Zip_Code = Zip
...
>>> Addr = Address(7,"High St","Anytown","123 456")
>>> print( Addr.HsNumber, Addr.Street )
```

Here is the explanation:

```
>>> class Address:
```

The `class` statement tells Python that we are about to define a new type called, in this case, `Address`. The colon indicates that any indented lines following will be part of the class definition. The definition will end at the next unindented line. If you are using IDLE you should find that the editor has indented the next line for you, if working at a command line Python prompt in an MS DOS window then you will need to manually indent the lines as shown. Python doesn't care how much you indent by, just so long as it is consistent.

```
...   def __init__(self, Hs, St, Town, Zip):
```

The first item within our class is what is known as a *method definition*. One very important detail is that the name has a double underscore at each end, this is a Python convention for names that it treats as having special significance. This particular method is called `__init__` and is a special operation, performed by Python, when we create an instance of our new class, we'll see that shortly. The colon, as before, simply tells Python that the next set of indented lines will be the actual definition of the method.

```
...     self.HsNumber = Hs
```

This line plus the next three, all assign values to the internal fields of our object. They are indented from the `def` statement to tell Python that they constitute the actual definition of the `__init__` operation.The blank line tells the Python interpreter that the class definition is finished so that we get the `>>>` prompt back.

```
>>> Addr = Address(7,"High St","Anytown","123 456")
```

This creates a new instance of our Address type and Python uses the `__init__` operation defined above to assign the values we provide to the internal fields. The instance is assigned to the `Addr` variable just like an instance of any other data type would be.

```
>>> print( Addr.HsNumber, Addr.Street )
```

Now we print out the values of two of the internal fields using the dot operator to access them.

As I said we cover all of this in more detail later in the tutorial. The key point to take away is that Python allows us to create our own data types and use them pretty much like the built in ones.

Points to remember

- Variables refer to data and may need to be declared before being defined.
- Data comes in many types and the operations you can successfully perform will depend on the type of data you are using.
- Simple data types include character strings, numbers, Boolean or 'truth' values.
- Complex data types include collections, files, dates and user defined data types.
- There are many operators in every programming language and part of learning a new language is becoming familiar with both its data types and the operators available for those types.
- The same operator (e.g. addition) may be available for different types, but the results may not be identical, or even apparently related!

Previous Next Contents

If you have any questions or feedback on this page send me mail at: alan.gauld@btinternet.com

More Sequences and Other Things

What will we cover?
• We introduce a new tool for entering Python programs. • We review the use of variables to store information until we need it. • We discuss comments and why they are needed • We combine longer sequences of commands to perform a task.

OK, Now we know how to type simple, single entry, commands into Python and have started to consider data and what we can do with it. In doing so we typed in a few longer sequences of 5-10 lines. We are getting close to being able to write really quite useful programs but with one big snag: every time we exit Python we lose our programs. If you have been doing the VBScript or JavaScript examples you will see that you have stored those examples in files and so can run them repeatedly, we need to do the same with Python. I already mentioned that we can do this using any text editor, like notepad or pico, say, and saving the file with a .py file extension. You can then run the file from the operating system command prompt by prefixing the script name with `python`. Here is an example that uses a sequence of python commands; all things we've seen already:

```
#  File: firstprogram.py
print( "hello world" )
print( "Here are the ten numbers from 0 to 9\n0 1 2 3 4 5 6 7 8 9" )
print( "I'm done!" )
# end of file
```

Note that the lines at top and bottom are not really needed, they are called *comments* and we will discuss them later in this topic. I added them to show more clearly what goes into the file. You can use Notepad or any other text editor to create the file so long as it saves in plain text.

Now to run this program start up an operating system command prompt. (If you aren't sure about how to do that then see the box in the Getting Started topic.) Change into the directory where you saved your python file and execute it by prefixing its name with python, like this:

```
D:\PROJECTS\Python> python firstprogram.py
hello world
Here are the ten numbers from 0 to 9
0 1 2 3 4 5 6 7 8 9
I'm done!

D:\PROJECTS\Python>
```

You can see the Windows command prompt and the command I typed (in bold), plus the output of the program displayed before the command prompt reappears.

However, there is an easier way...

The joy of being IDLE

When you installed Python you also installed a useful application, itself written in Python, called IDLE. IDLE is what is known as an *Integrated Development Environment*, that is to say it includes several tools that help the programmer, all wrapped up in a single application. I won't be looking at IDLE in depth here, but the two features that I want to highlight are the fact that it provides an enhanced version of the Python >>> prompt, complete with *syntax highlighting* (That is, displaying

different features of the language in different colours) and other nice features, plus a nice, Python specific, text editor which allows you to run your program files (such as the one we created above) directly from within IDLE.

I strongly recommend that, if you haven't already done so, you give IDLE a try. The best place to start, once you find the right way to start IDLE on your Operating System, is to visit Danny Yoo's excellent tutorial.

There is also a full tutorial on using IDLE on the Python web site under the IDLE topic.

Finally, if you prefer a simple approach, you can find several text editors that support programming in Python in various ways. The vim editor provides syntax highlighting (colouring of key words etc), emacs has a full editing mode for Python and Scite is a very lightweight editor that provides Python syntax highlighting and other nice features.

If you go down the text editor route you will likely find it most convenient to have three windows open on your screen at once:

1. The editor where you type in and save your source code
2. A Python session where you try things out at the >>> prompt before adding them to your program in the editor and
3. An operating system command prompt used to run the program to test it.

I personally prefer the 3 window approach, but most beginners seem to prefer the all-in-one style of an IDE like IDLE. The choice is entirely up to you.

If you are using JavaScript or VBScript I recommend using one of the editors mentioned above and a suitable web browser, say Internet Explorer, opened at the file you are working on. To test changes just hit the Reload button in the browser.

A quick comment

One of the most important of programming tools is one that beginners often feel is useless on first acquaintance - comments. Comments are just lines in the program which describe what's going on. They have no effect whatsoever on how the program operates, they are purely decorative. They do, however, have an important role to play - they tell the programmer what's going on and more importantly *why*. This is especially important if the programmer reading the code isn't the one who wrote it, or, it's a long time since he/she wrote it. Once you've been programming for a while you'll really appreciate good comments. I have actually been adding comments to some of the code fragments that you've seen already, they were the green bits of the lines with a # (Python) or ' (VBScript) symbol in front of them. From now on I'll be commenting the code fragments that I write. Gradually the amount of explanatory text will diminish as the explanation appears in comments instead.

Every language has a way of indicating comments. In VBScript it's `REM` (for Remark) or, more commonly, a single quote `'` at the beginning of a comment. Everything after the marker is ignored:

```
REM This never gets displayed
' neither does this
msgBox "This gets displayed"
```

You might recognize REM if you have ever written any MSDOS batch files, since they use the same comment marker.

Note that the use of a single quote as a comment marker is the reason you can't start a string with a single quote in VBScript - VBScript thinks it's a comment!

Python uses a `#` symbol as its comment marker. Anything following a `#` is ignored:

```
v = 12     # give v the value 12
x = v*v    # x is v squared
```

Incidentally this is very *bad* commenting style. Your comment should not merely state what the code does - we can see that for ourselves! It should explain *why* it's doing it:

```
v = 3600    # 3600 is num of secs in an hour
s = t*3600  # t holds elapsed time in hours, so convert to secs
```

These are much more helpful comments.

Finally JavaScript uses a double slash: `//` as a comment marker. Once again, everything after the marker gets ignored.

Some languages allow multi-line comments between a pair of markers, but this can lead to some obscure faults if the terminating marker is not correctly input. JavaScript allows multi-line comments by using the pair of markers: `/*` followed by `*/`, like this:

```
<script type="text/javascript">
document.write("This gets printed\n");

// A single line comment

/* Here is a multi line comment. It continues from this line
down into this line and even
onto this third line. It does not appear in the script output.
It is terminated by a mirror image of the opening marker */

document.write("And this prints too");
</script>
```

The important point about comments is that they are there to explain the code to anyone who tries to read it. With that in mind you should explain any mysterious sections - such as apparently arbitrary values used, or complex arithmetic formulae etc. And remember, the puzzled reader might be yourself in a few weeks or months time!

Sequences using variables

We introduced the concept of variables in the Raw Materials topic topic. There we said they were labels with which we marked our data for future reference. We saw some examples of using variables too in the various list and address book examples. However variables are so fundamentally important in programming that I want to do a quick recap of how we use variables before moving onto new things.

Now, at a Python Prompt(>>>), either in IDLE's Shell or in a DOS or Unix command window, try typing this:

```
>>> v = 7
```

```
>>> w = 18
>>> x = v + w    # use our variables in a calculation
>>> print( x )
```

What's happening here is that we are creating variables (`v, w, x`) and manipulating them. It's rather like using the `M` button on your pocket calculator to store a result for later use.

We can make this prettier by using a format string to print the result:

```
>>> print( "The sum of %d and %d is: %d" % (v,w,x) )
```

One advantage of format strings is that we can store them in variables too:

```
>>> s = "The sum of %d and %d is: %d"
>>> print( s % (v,w,x) )   # useful if printing same output with different values
```

This makes the print statement much shorter, especially when it contains many values. However it also makes it more cryptic so you have to use your judgment to decide whether very long lines are more or less readable than a stored format value. If you keep the format string beside the print statement, as we did here, then it's not too bad. Finally one other thing that helps is to name your variables in such a way that they explain what they are used for. For example instead of calling the format string `s` I could have called it `sumFormat`, so that the code looked like this:

```
>>> sumFormat = "The sum of %d and %d is: %d"
>>> print( sumFormat % (v,w,x) )   # useful if printing same output with differen
```

Now, in a program with several different format strings in use, we could more easily tell which format is being printed. Meaningful variable names are always a good idea and I'll try to use meaningful names where possible. Up until now our variables haven't had much meaning to convey!

Order matters

By now you might be thinking that this sequence construct is a bit over-rated and obvious. You would be right in so far as it's fairly obvious, but it's not quite as simple as it might seem. There can be hidden traps. Consider the case where you want to 'promote' all the headings in an HTML document up a level:

Now in HTML the headings are indicated by surrounding the text with
`<H1>`text`</H1>` for level 1 headings,
`<H2>`text`</H2>` for level 2 headings,
`<H3>`text`</H3>` for level 3 headings and so on.

The problem is that by the time you get to level 5 headings the heading text is often smaller than the body text, which looks odd. Thus you might decide to promote all headings up one level. It's fairly easy to do that with a simple string substitution in a text editor, substitute '<H2' with '<H1' and '</H2' with '</H1' and so on.

Consider though what happens if you start with the highest numbers - say H4 -> H3, then do H3 -> H2 and finally H2 -> H1. All of the headings will have moved to H1! Thus the order of the sequence of actions is important. The same is just as true if we wrote a program to do the substitution (which we might well want to do, since promoting headings may be a task we do regularly).

We've seen several other examples using variables and sequences in the Raw Materials topic - particularly the various address book examples. Why not think up a few examples for yourself? Once you've done that we'll move on to a case study that we will build upon as we move through the tutorial, improving it with each new technique we learn.

A Multiplication Table

I'm now going to introduce a programming exercise that we will develop over the next few chapters. The solutions will gradually improve as we learn new techniques.

Recall that we can type long strings by enclosing them in triple quotes? Let's use that to construct a multiplication table:

```
>>> s = """
1 x 12 = %d
2 x 12 = %d
3 x 12 = %d
4 x 12 = %d
"""     # be careful - you can't put comments inside
>>>     # strings, they'll  become part of the string!
>>> print( s % (12, 2*12, 3*12, 4*12) )
```

By extending that we could print out the full 12 times table from 1 to 12. But is there a better way? The answer is yes, let's see what it is.

Points to remember
• IDLE is a cross platform development tool for writing Python programs. • Comments can make programs clearer to read but have no effect on the operation of the program • Variables can store intermediate results for later use

Previous Next Contents

If you have any questions or feedback on this page send me mail at: alan.gauld@btinternet.com

Looping - Or the art of repeating oneself!

What will we cover?
• How to use loops to cut down on repetitive typing. • Different types of loop and when to use them.

In the last exercise we printed out part of the 12 times table. But it took a lot of typing and if we needed to extend it, it would be very time consuming. Fortunately there is a better way and it's where we start to see the real power that programming languages offer us.

FOR Loops

What we are going to do is get the programming language to do the repetition, substituting a variable which increases in value each time it repeats. In Python it looks like this:

```
>>>for n in range(1,13):
...     print( "%d x 12 = %d" % (n, n*12) )
...
1 x 12 = 12
2 x 12 = 24
3 x 12 = 36
4 x 12 = 48
5 x 12 = 60
6 x 12 = 72
7 x 12 = 84
8 x 12 = 96
9 x 12 = 108
10 x 12 = 120
11 x 12 = 132
12 x 12 = 144
```

Note 1: The for line ends with a colon (`:`). This is important since it signifies to Python that what follows is the thing to be repeated.

Note 2: We need the `range(1,13)` to specify 13 because `range()` function generates from the first number up to, but not including, the second number. This may seem somewhat bizarre at first but there are reasons and you get used to it.

Note 3: The `for` operator in Python is actually a *foreach* operator in that it applies the subsequent code sequence to each member of a collection. In this case the collection is the list of numbers generated by `range()`. You can prove that by typing `print( list( range(1,13) )` at the python prompt and seeing what gets printed. Alternatively we could just replace `range()` with an explicit list of numbers like this:

```
>>> for n in [1,2,3,4,5,6,7,8,9,10,11,12]:
...     print( "%d x 12 = %d" % (n, n*12) )
...
```

Note 4: The `print` line is *indented* or spaced further in than the `for` line above it. That is a very important point since it's how Python knows that the `print` is the bit to repeat. There can be more than a single line indented too, Python will repeat all of the lines that are indented for each item in the collection. Also, it doesn't matter how much indentation you use so long as it's consistent.

Note 5: In the interactive interpreter you need to hit return twice to get the program to run. The reason is that the Python interpreter can't tell whether the first one is another line about to be added to the loop code or not. When you hit Enter a second time Python assumes your finished entering code and runs the program.

Phew! That was a lot of notes! However now we have considered the syntax of a for loop let's consider how it works. Let's step through it step by step.

First of all, python uses the range() function to create a list of numbers from 1 to 12.

Next python makes `n` equal to the first value in the list, in this case 1. It then executes the bit of code that is indented, using the value n = 1:

```
   print( "%d x 12 = %d" % (1, 1*12) )
```

Python then goes back to the `for` line and sets `n` to the next value in the list, this time 2. It again executes the indented code, this time with n = 2:

```
   print( "%d x 12 = %d" % (2, 2*12) )
```

It keeps repeating this sequence until it has set `n` to all the values in the list. At that point it moves to the next command that is **not** indented - in this case there aren't any more commands so the program stops.

Here's the same loop in VBScript:

The simplest VBScript loop construct is called a `For...Next` loop, and is used as shown:

```
<script type="text/vbscript">
For N = 1 To 12
    MsgBox N & " x 12 = " & N*12
Next
</script>
```

This is much more explicit and easier to see what is happening. The value of `N` varies from 1 through to 12 and the code before the `Next` keyword is executed. In this case it just prints the result in a dialog box as we've seen before. The indentation is optional but makes the code easier to read.

Although the VBScript appears, at first glance, more obvious, the Python version is ultimately more flexible as we'll see shortly.

And in JavaScript

JavaScript uses a `for` construct that is common in many programming languages, being modeled on C. It looks like this:

```
<script type="text/javascript">
for (n=1; n <= 12; n++){
    document.write(n + " x 12 = " + n*12 + "<BR>");
    };
</Script>
```

Note: This construct initially looks quite complicated. It has 3 parts inside the parentheses:

- an *initializing* part: `n = 1` executed just once, before anything else,
- a *test* part: `n <= 12` which is executed before each iteration and
- an *increment* part: `n++` which is shorthand for "increment n by 1", and is executed after each iteration.

Notice also that JavaScript encloses the repeated code (the *loop body*) in braces `{}` and although that is all that is needed, technically speaking, it is considered good practice to indent the code inside the braces too, just to improve readability.

The *loop body* will only execute if the *test* part is true. Each of these parts can contain arbitrary code but the test part must evaluate to a boolean value.

More about the Python *for* construct

The Python `for` loop iterates over a sequence. A Sequence in Python, lest you forgot, includes things like strings, lists and tuples. (In fact Python can iterate over several other kinds of things too but we will discuss them much later in the tutorial.) Thus we can write `for` loops that act on any type of sequence. Let's try printing the letters of a word one by one using a `for` loop with a string:

```
>>> for c in 'word': print( c )
...
w
o
r
d
```

Notice how the letters were printed, one per line. Notice too that where the body of the loop consists of a single line we can add it on the same line after the colon(:). The colon is what tells Python that there's a block of code coming up next.

We can also iterate over a tuple:

```
>>> for word in ('one','word', 'after', 'another'): print (word)
...
```

This time we got each word on a line. We can put them all on one line using a special feature of the `print()` function. We can add an extra *argument* after the printable item, like this:

```
>>> for word in ('one', 'word', 'after', 'another'): print( word, end='' )
...
```

See how the words now appear as a single line? The `end=''` part told Python to use an empty string (`''` as the line ending instead of the newline character that it uses by default.

We have already seen `for` with a list but for completeness we will do it once more:

```
>>> for item in ['one', 2, 'three']: print( item )
...
```

There is one caveat when using foreach style loops like this. The loop gives you a copy of what was in the collection, you can't modify the contents of the collection directly. So if you need to modify the collection you have to use an awkward kludge involving the index of the collection, like this:

```
myList = [1,2,3,4]
for index in range(len(myList)):
    myList[index] += 1
print( myList )
```

That will increment each entry in `myList`. If we had not used the index trick we would simply have incremented the copied items but not changed the original list.

Note that in this example I have not used the interactive Python prompt (>>>), so you need to type this into a file as described in the More sequences topic. If you do try typing it at the >>> prompt you will need to add extra blank lines to tell Python when you finish a block, for example after the `myList =` line. It's actually quite a good way of learning where blocks start and stop: to type the code in and see if you correctly guess where an extra line will be needed. It should be where the indentation changes!

The other gotcha with for loops is that you can't delete items from the collection that you are iterating over, otherwise the loop will get confused. It's a bit like the old cartoon character cutting off the branch of a tree while sitting on it! The best way to deal with this situation is to use a different kind of loop, which we are going to discuss next. However to understand how to remove elements safely we need to wait until we cover yet another topic, that of branching, so we will explain this subject when we get there.

It's worth noting that VBScript and JavaScript each have loop constructs for looping over the elements in a collection. I won't discuss them in detail here, but the VBScript construct is `for each...in...` and the JavaScript version is `for...in...` You can look them up in the relevant help pages if you want to see the details.

WHILE Loops

`FOR` loops are not the only type of looping construct available. Which is just as well, since `FOR` loops require us to know, or be able to calculate in advance, the number of iterations that we want to perform. So what happens when we want to keep doing a specific task until something happens but we don't know when that something will be? For example, we might want to read and process data from a user until the user tells us to stop, so we won't know in advance how many data items the user wants to have processed. We just want to keep on processing data until the user says enough. That's possible but kind of tricky, in a `FOR` loop.

To solve this problem we have another type of loop: the *WHILE* loop.

It looks like this in Python:

```
>>> j = 1
>>> while j <= 12:
...     print( "%d x 12 = %d" % (j, j*12) )
...     j = j + 1
```

Let's walk through what's happening.

1. First we initialize `j` to 1, initializing the *control variable* of a while loop is a very important first step, and a frequent cause of errors when missed out.
2. Next we execute the `while` statement itself, which evaluates a *boolean expression*
3. If the result is True it proceeds to execute the indented block which follows. In our example `j` is less than 12 so we enter the block.

4. We execute the print statement to output the first line of our table.
5. The next line of the block increments the control variable, `j`. In this case it's the last indented line, signifying the end of the while block.
6. We go back up to the `while` statement and repeat steps 4-6 with our new value of `j`.
7. We keep on repeating this sequence of actions until `j` reaches 13.
8. At that point the `while` test will return False and we skip past the indented block to the next line with the same indentation as the `while` statement.
9. In this case there are no other lines so the program stops.

By now that should feel pretty straightforward. Just one thing to point out - do you see the colon (:) at the end of the `while` (and `for`) lines above? That just tells Python that there's a chunk of code (a *block*) coming up. As we'll see in a moment, other languages have their own ways of telling the interpreter to group lines together, Python uses a combination of the colon and indentation.

VBScript

Let's look at VBScript's version of the while loop:

```
<script type="text/vbscript">
DIM J
J = 1
While J <= 12
    MsgBox J & " x 12 = " & J*12
    J = J + 1
Wend
</script>
```

This produces the same result as before but notice that the loop block is delimited by the keyword `Wend` (short for While End obviously!). Other than that it works pretty much exactly like the Python one.

JavaScript

```
<script type="text/javascript">
j = 1;
while (j <= 12){
   document.write(j," x 12 = ",j*12,"<BR>");
   j = j++;
   }
</script>
```

As you see the structure is pretty similar just some curly brackets or *braces* instead of the `Wend` in VBScript. (Remember that j++ in JavaScript means increment the value of j) Note that unlike Python, neither VBScript nor JavaScript need any indentation, that's purely to make the code more readable.

Finally it's worth comparing the JavaScript `for` and `while` loops. Recall that the `for` loop looked like this:

```
for (j=1; j<=12; j++){....}
```

Now, that is exactly the same structure as the `while` loop, just compressed into one line. The initializer, the test condition and the loop modifier are all there clearly seen. So in fact a JavaScript `for` loop is simply a `while` loop in a more compact form. It would be possible to do without the `for` loop completely and only have `while` loops, and that's exactly what some other languages do.

More Flexible Loops

Coming back to our 12 times table at the beginning of this section. The loop we created is all very well for printing out the 12 times table. But what about other values? Can you modify the loop to make it do the 7 times table say? It should look like this:

```
>>> for j in range(1,13):
...     print( "%d x 7 = %d" % (j,j*7) )
```

Now this means we have to change the 12 to a 7 twice. And if we want another value we have to change it again. Wouldn't it be better if we could enter the multiplier that we want?

We can do that by replacing the values in the print string with another variable. Then set that variable before we run the loop:

```
>>> multiplier = 12
>>> for j in range(1,13):
...     print( "%d x %d = %d" % (j, multiplier, j*multiplier) )
```

That's our old friend the 12 times table. But now to change to the seven times, we only need to change the value of 'multiplier'. Try typing this program into a Python script file and running it from a command prompt. Then edit the multiplier value to try out some different tables.

Notice that we have here combined sequencing and loops. We have first a single command, `multiplier = 12` followed, *in sequence* by a `for` loop.

Looping the loop

Let's take the previous example one stage further. Suppose we want to print out all of the times tables from 2 to 12 (1 is too trivial to bother with). All we really need to do is set the multiplier variable as part of a loop, like this:

```
>>> for multiplier in range(2,13):
...     for j in range(1,13):
...         print( "%d x %d = %d" % (j,multiplier,j*multiplier) )
```

Notice that the part indented inside the first `for` loop is exactly the same loop that we started out with. It works as follows:

1. We set multiplier to the first value (2) then go round the second, inner loop.
2. Then we set multiplier to the next value (3) and go round the inner loop again,
3. and so on.

This technique is known as *nesting* loops.

One snag is that all the tables merge together, we could fix that by just printing out a separator line at the end of the first loop, like this:

```
>>> for multiplier in range(2,13):
...     for j in range(1,13):
...         print "%d x %d = %d" % (j,multiplier,j*multiplier)
...     print( "------------------- " )
```

Note that the second print statement lines up with the second 'for', it is the second statement in the loop sequence. Remember, the indenting level is very important in Python.

Just for comparisons sake let's see how that looks in JavaScript too:

```
<script type="text/javascript">
for (multiplier=2; multiplier < 13; multiplier++){
    for (j=1; j <= 12 ; j++){
        document.write(j, " x ", multiplier, " = ", j*multiplier, "<BR>");
        }
    document.write("---------------<BR>");
    }
</script>
```

Experiment with getting the separator to indicate which table it follows, in effect to provide a caption. Hint: You probably want to use the multiplier variable and a Python format string.

Other loops

Some languages provide more looping constructs but some kind of `for` and `while` are usually there. (Modula 2 and Oberon only provide `while` loops since while loops can simulate `for` loops - as we saw above.) Other loops you might see are:

do-while
Same as a while but the test is at the end so the loop always executes at least once.

repeat-until
Similar to above but the logic of the test is reversed.

GOTO, JUMP, LOOP etc
Mainly seen in older languages, these usually set a marker in the code and then explicitly jump directly to that marker.

Points to remember
• `FOR` loops repeat a set of commands for a fixed number of iterations. • `WHILE` loops repeat a set of commands until some terminating condition is met. They may never execute the *body* of the loop if the terminating condition is false to start with. • Other types of loops exist but `FOR` and `WHILE` are nearly always provided. • Python `for` loops are really `foreach` loops - they operate on a list of items. • Loops may be nested one inside another.

Previous Next Contents

If you have any questions or feedback on this page send me mail at: alan.gauld@btinternet.com

Coding Style

What will we cover?
• Several new uses for comments • How to layout code using indentation to improve readability • An introduction to the use of modules for storing our programs

Comments

I've already spoken about comments in the 'More Sequences' section. However there are some other things we can do with comments and I'll enlarge on those here:

Version history information

It is good practice to create a file header at the start of each file. This should provide details such as the creation date, author, date of last change, version and a general description of the contents. Often a log of changes. This block will appear as a comment:

```
##############################
# Module:   Spam.py
# Author:   A.J.Gauld
# Date:     1999/09/03
# Version:  Draft 0.4
'''
This module provides a Spam class which can be
combined with any other type of Food object to create
interesting meal combinations.
'''
################################
# Log:
# 1999/09/01    AJG - File created
# 1999/09/02    AJG - Fixed bug in pricing strategy
# 1999/09/02    AJG - Did it right this time!
# 1999/09/03    AJG - Added broiling method(cf Change Req #1234)
################################
import sys, string, food
...
```

Thus when you first open a file it should contain a nice summary of what the file is for, what's changed over time and who did it and when. This is particularly important if you are working on a team project and need to know who to ask about the design or the changes. There are version control tools available that can help automate the production of some of this documentation, but they are outside the scope of this tutorial.

Note that I put the description in between two sets of triple quotes. This is a Python specific trick known as a *documentation string* that makes the description available to Python's built-in `help()` function as we'll see shortly.

It is also worth noting that there are source code repository tools which can automatically maintain things like the author, filename, and version log details. Once you start using a source code repository (such as SCCS, CVS, Subversion or ClearCase) it is worth taking the time to investigate those features as they can eliminate a lot of clerical administration of comments.

Commenting out redundant code

This technique is often used to isolate a faulty section of code. For example, assume a program reads some data, processes it, prints the output and then saves the results back to the data file. If the results are not what we expect it would be useful to temporarily prevent the (erroneous)data being saved back to the file and thus corrupting it. We could simply delete the relevant code but a less radical approach is simply to convert the lines into comments like so:

```
data = readData(datafile)
for item in data:
    results.append(calculateResult(item))
printResults(results)
######################
# Comment out till bug in calculateResult fixed
# for item in results:
#     dataFile.save(item)
######################
print 'Program terminated'
```

Once the fault has been fixed we can simply delete the comment markers to make the code active once more. Some editing tools, including IDLE, have menu options to comment out a selected block of code, and to uncomment it later.

Note that many programmers editors, including IDLE, have a feature whereby you can select a section of code and get the editor to comment it out automatically and then uncomment it when done. This is is the *Format->Comment Out Region* menu item in IDLE.

Documentation strings

All languages allow you to create comments to document what a function or module does, but a few, such as Python and Smalltalk, go one stage further and allow you to document the function in a way that the language/environment can use to provide interactive help while programming. In Python this is done using the `"""documentation"""` string style:

```
class Spam:
    """A meat for combining with other foods

    It can be used with other foods to make interesting meals.
    It comes with lots of nutrients and can be cooked using many
    different techniques"""

    def __init__(self):
       pass   # ie. it does nothing!

help(Spam)
```

Note: We can access the *documentation string* by using the `help()` function. Modules, Functions and classes/methods can all have documentation strings. For example try:

```
>>> import sys
>>> help (sys.exit)
Help on built-in function exit:

exit(...)
    exit([status])
```

```
    Exit the interpreter by raising SystemExit(status).
    If the status is omitted or None, it defaults to zero (i.e., success).
    If the status is numeric, it will be used as the system exit status.
    If it is another kind of object, it will be printed and the system
    exit status will be one (i.e., failure).
(END)
```

To get out of help mode hit the letter 'q'(for quit) when you see the `(END)` marker. If more than one page of help is present you can hit the space bar to page through it. If you are using IDLE, or other IDE, then you likely won't see the `(END)` marker rather it will simply display all the text and you need to use the scroll bars to go back and read it.

One final helper function is `dir()` which displays all the features that Python knows about for a particular object. Thus if you want to know what functions or variables are contained in the sys module, for example you could do this:

```
>>> import sys
>>> dir(sys)
[..... 'argv', 'builtin_module_names', 'byteorder', .... 'copyright',
.... 'exit', ..... 'stderr', 'stdin', 'stdout', 'subversion',
'version', 'version_info', 'warnoptions', 'winver']
```

You can then select likely candidates and use help() to get more details. (Note, I have missed out many of the entries to save space!) This is particularly useful if you are using a module that does not have good documentation (or even has no documentation!)

Block Indentation

This is one of the most hotly debated topics in programming. It almost seems that every programmer has his/her own idea of the best way to indent code. As it turns out there have been some studies done that show that at least some factors are genuinely important beyond cosmetics - ie they actually help us understand the code better.

The reason for the debate is simple. In most programming languages the indentation is purely cosmetic, an aid to the reader. (In Python it is, in fact, needed and is essential to proper working of the program!) Thus:

```
< script type="text/vbscript">
For I = 1 TO 10
    MsgBox I
Next
</script>
```

Is exactly the same as:

```
< script type="text/vbscript">
For I = 1 TO 10
MsgBox I
Next
</script>
```

so far as the VBScript interpreter is concerned. It's just easier for us to read with indentation.

The key point is that indentation should reflect the logical structure of the code thus visually it should follow the flow of the program. To do that it helps if the blocks look like blocks thus:

```
XXXXXXXXXXXXXXXXXXX
    XXXXXXXXXXXXXXX
    XXXXXXXXXXXXXXX
    XXXXXXXXXXXXXXX
```

which reads better than:

```
XXXXXXXXXXXXXXXXXXXX
  XXXXX
    XXXXXXXXXXXX
    XXXXXXXXXXXX
    XXXXXXXXXXXX
  XXXXX
```

because it's clearly all one block. Studies have shown significant improvements in comprehension when indenting reflects the logical block structure. In the small samples we've seen so far it may not seem important but when you start writing programs with hundreds or thousands of lines it will become much more so.

Variable Names

The variable names we have used so far have been fairly meaningless, mainly because they had no meaning but simply illustrated techniques. In general it's much better if your variable names reflect what you want them to represent. For example in our times table exercise we used 'multiplier' as the variable to indicate which table we were printing. That is much more meaningful than simply 'm' - which would have worked just as well and been less typing.

Its a trade-off between comprehensibility and effort. Generally the best choice is to go for short but meaningful names. Too long a name becomes confusing and is difficult to get right consistently (for example I could have used `the_table_we_are_printing` instead of `multiplier` but it's far too long and not really much clearer.

Saving Your Programs

While the Python interactive interpreter prompt (>>>) is very useful for trying out ideas quickly, it loses all you type the minute you exit. In the longer term we want to be able to write programs and then run them over and over again. To do this in Python we create a text file with an extension `.py` (this is a convention only, you could use anything you like. But it's a good idea to stick with convention in my opinion...). You can then run your programs from an Operating System command prompt by typing:

```
C:\WINDOWS> python spam.py
```

Where `spam.py` is the name of your Python program file and the `C:\WINDOWS>` is the operating system prompt.

If you did follow convention you can also start your programs by double clicking them in Windows Explorer since Windows knows to associate the `.py` extension with the Python interpreter.

The other advantage of using files to store the programs is that you can edit mistakes without having to retype the whole fragment or, in IDLE, cursor all the way up past the errors to reselect the code. IDLE supports having a file open for editing and running it from the *Run->Run module* menu item (or the F5 keyboard shortcut).

From now on I won't normally be showing the `>>>` prompt in examples, I'll assume you are creating the programs in a separate file and running them either within IDLE or from a command prompt (my personal favourite).

Note for Windows users

Under Windows you can set up a file association for files ending `.py` within Explorer. This will allow you to run Python programs by simply double clicking the file's icon. This should already have been done by the installer. You can check by finding some .py files and trying to run them. If they start (even with a Python error message) it's set up. (The icon should be the Python logo.) The problem you will likely run into at this point is that the files will run in a DOS box and then immediately close, so fast you scarcely even see them! There are a couple of options:

- The first way is simplest and involves putting the following line of code at the end of each program:

  ```
  input("Hit ENTER to quit")
  ```

  Which simply displays the message and waits for the user to hit the ENTER or Return key. We will discuss `input()` in the next topic.

- The second technique uses the Windows Explorer settings. The procedure is fairly standard but may vary according to the version of Windows you have. I will describe Windows XP Home.
  1. First select a `.py` file and go to the `Tools->Folder Options` menu item.
  2. In the dialog box select the `File Types` tab.
  3. Scroll down till you find the `PY` file type and click on it to select it.
  4. Click the `Advanced` button at the bottom.
  5. In the new dialog select `open` from the list and click `Edit...`
  6. In the new dialog you should see the `Application...` line say something like:

     ```
     C:\PYTHON3\python.exe "%1" %*
     ```

     Edit it to add a `-i` after the `python.exe` and before the `%1`, like this:

     ```
     C:\PYTHON3\python.exe -i "%1" %*
     ```
  7. Now close all the dialogs.

  This will stop Python from exiting at the end of your program and leave you at the >>> prompt where you can inspect variable values etc, or just exit manually. (An alternative trick, which I prefer, is, at step 5, to add a new option called `Test` alongside the existing `Open`. Set the Test option to the command line above, complete with `-i`. This allows you to Right Click in Explorer and choose `Open` to run the program and have it close automatically or choose `Test` to run the program finishing in Python. The choice of behaviour is then yours.)

Note for Unix users

The first line of a Python *script* file should contain the sequence `#!` followed by the full path of python on your system. (This is sometimes known as the *shebang* line.) You can find that by typing, at your shell prompt:

```
$ which python
```

On my system the line looks like:

```
#! /usr/local/bin/python
```

This will allow you to run the file without calling Python at the same time (after you set it to be executable via chmod - but you knew that already I'm sure!):

```
$ spam.py
```

You can use an even more convenient trick on most modern Unix systems (including all Linux distros) which replaces the path information with `/usr/bin/env/python`, like this:

```
#! /usr/bin/env/python
```

That will find where Python is in your path automatically. The only snag is where you may have two or more different versions of Python installed and the script will only work with one of them (maybe it uses a brand new language feature, say), in that case you will be better with the full path technique.

This `#!` line doesn't do any harm under Windows/Mac either, since it just looks like a comment, so those users can put it in too, if their code is likely to ever be run on a unix box.

VBScript & JavaScript

You VBScript and JavaScript users can ignore the above, you've already been saving your programs as files, it's the only way to get them to work!

Points to remember
• Comments can be used to temporarily prevent code from executing, which is useful when testing or 'debugging' code. • Comments can be used to provide an explanatory header with version history of type file. • Documentation strings can be used to provide run-time information about a module and the objects within it. • Indentation of blocks of code helps the reader see clearly the logical structure of the code. • By typing a python program into a file instead of at the Python '>>>' prompt the program can be saved and run on demand by typing `$ python progname.py` at the command prompt or by double clicking the filename within an Explorer window on Windows.

Previous Next Contents

If you have any questions or feedback on this page send me mail at: alan.gauld@btinternet.com

Conversing with the user

What will we cover?
• How to prompt the user to enter data and how to read that data once it is entered. • We will show how to read both numerical and string based data. • The concepts of *stdin* and *stdout* • We look at command line interfaces and how to read data input as command line arguments. • We discuss the EasyGUI module for displaying simple data entry dialog boxes

So far our programs have only dealt with static data. Data that, if need be, we can examine before the program runs and thus write the program to suit. Most programs aren't like that. Most programs expect to be driven by a user, at least to the extent of being told what file to open, edit etc. Others prompt the user for data at critical points. This aspect of programming is what is referred to as the *User Interface* and in commercial programs designing and building the user interface is a job for specialists trained in human machine interaction and ergonomics. The average programmer does not have that luxury so must make do with some common sense, and careful thought about how users will use the program. The most basic feature of a User Interface is displaying output and we have already covered the most primitive way of doing that via the Python `print` function (and JavaScript's `write()` function as well as the VBScript `MsgBox` dialog). The next step in User Interface design is to take input directly from the user. The simplest way to do that is for the program to ask for the input at run time, the next simplest way is for the user to pass the data in when he or she starts the program, finally we have *graphical user interfaces (GUIs)* with text entry boxes etc. In this topic we look mainly at the first two methods. We introduce GUI programming much later in the tutor because it is significantly more complex, however there is a module which allows us to do very basic GUI style dialog boxes for data entry in Python and we will briefly consider that.

Let's see how we can get data from a user in a normal Python interactive session running in IDLE or an OS terminal. Afterwords we'll try doing the same in a program.

Python user input

We can get input from a user in Python like this:

```
>>>> print( input("Type something: ") )
```

As you see `input()` simply displays the given *prompt* - "Type something" in this case - and captures whatever the user types in response. `print()` then displays that response. We could instead assign it to a variable:

```
>>> resp = input("What's your name? ")
```

And now we can print out whatever value was captured:

```
>>> print( "Hi " + resp + ", nice to meet you" )
```

Notice that this time I have chosen not to use the string formatting operator to display the value stored in the variable `resp` and have instead just inserted the value between two strings joining all three strings together using the string addition operator. The value of `resp` is the one captured from the user by `input()`.

Notice too in both examples the use of spaces inside the strings, both in the prompt given to `input()` but also in the output string. In particular notice the third part of the output string started with a comma followed by a space. It is a common mistake when producing output like that to get the spacing wrong so check carefully when testing your programs.

This is great for reading strings. But what about other data types? The answer is that Python comes with a full set of data conversion functions that can convert a string to another data type. Obviously the data in the string has to be compatible with the type, otherwise you will get an error!

As an example lets take our multiplication table example and modify it to read the multiplier value from the user:

```
>>> multiplier = input("Which multiplier do you want? Pick a number ")
>>> multiplier = int(multiplier)
>>> for j in range(1,13):
...   print( "%d x %d = %d" % (j, multiplier, j * multiplier) )
```

Here we read the value from the user then convert it to an integer using the `int()` conversion function. (You can use `float()` to convert to a floating point value too, should you need to). Here we did the conversion on a separate line to make it clearer what we were doing, but in practice this is so common that we usually just wrap the `input()` call inside the conversion, like this:

```
>>> multiplier = int( input("Which multiplier do you want? Pick a number ") )
>>> for j in range(1,13):
...   print( "%d x %d = %d" % (j, multiplier, j * multiplier) )
```

You see? We just wrapped the `input()` call inside the call to `int()`.

So what about using this in a real program? You recall the address book examples using a dictionary that we created in the raw materials topic? Let's revisit that address book now that we can write loops and read input data.

```
# create an empty address book dictionary
addressBook = {}

# read entries till an empty string
print()  # print a blank line
name = "-"  # non blank
while name != "":
   name = input("Type the Name(leave blank to finish): ")
   if name != "":
      entry = input("Type the Street, Town, Phone.(Leave blank to finish): ")
      addressBook[name] = entry

# now ask for one to display
name = "-"
while name != "":
   name = input("Which name to display?(leave blank to finish): ")
   if name != "":
      print( name, addressBook[name] )
```

That's our biggest program so far, and although the user interface design is a bit clunky it does the job. We will see how to improve it in a later topic. Some things to note in this program are the use of the boolean test in the `while` loops to determine when the user wants us to stop. Also note that whereas in the Raw Materials example we used a list to store the data as separate fields we have just

stored it as a single string here. That's because we haven't yet covered how to break down a string into separate fields. We'll cover that in a later topic too. In fact the address book program will be cropping up from time to time through the rest of the tutorial as we gradually turn it into something useful.

VBScript Input

In VBScript the InputBox statement reads input from the user thus:

```
<script type="text/vbscript">
Dim Input
Input = InputBox("Enter your name")
MsgBox ("You entered: " & Input)
</script>
```

The `InputBox` function simply presents a dialog with a prompt and an entry field. The contents of the entry field are returned by the function. There are various values that you can pass to the function such as a title string for the dialog box in addition to the prompt. If the user presses Cancel the function returns an empty string regardless of what is actually in the entry field.

Here is the VBScript version of our Address book example.

```
<script type="text/vbscript">
Dim dict,name,entry  ' Create some variables.
Set dict = CreateObject("Scripting.Dictionary")
name = InputBox("Enter a name", "Address Book Entry")
While name <> ""
   entry = InputBox("Enter Details - Street, Town, Phone number",
                    "Address Book Entry")
   dict.Add name, entry ' Add key and details.
   name = InputBox("Enter a name","Address Book Entry")
Wend

' Now read back the values
name = InputBox("Enter a name","Address Book Lookup")
While name <> ""
   MsgBox(name & " - " & dict.Item(name))
   name = InputBox("Enter a name","Address Book Lookup")
Wend
</script>
```

The basic structure is absolutely identical to the Python program although a few lines longer because of VBScript's need to pre-declare the variables with `Dim` and because of the need for a `Wend` statement to end each loop.

Reading input in JavaScript

JavaScript presents us with a challenge because it is a language primarily used within a web browser. We have a choice of using a simple input box like VBScript using the `prompt()` function or instead we can read from an HTML form element (or, in Internet Explorer, use Microsoft's Active Scripting technology to generate an `InputBox` dialog like the one used by VBScript). For variety I'll show you how to use the HTML form technique. If you are unfamiliar with HTML forms it might be worth finding an HTML reference or tutorial to describe them, alternatively just copy what I do here and hopefully it will be self explanatory. I will be keeping it very simple, I promise.

The basic structure of our HTML example will be to put the JavaScript code in a function, although we haven't covered these yet. For now just try to ignore the function definition bits.

```
<script type="text/javascript">
function myProgram(){
   alert("We got a value of " + document.entry.data.value);
}
</script>

<form name='entry'>
<P>Type value then click outside the field with your mouse</P>
<Input Type='text' Name='data' onChange='myProgram()'>
</form>
```

The program just consists of a single line that displays an `alert` box (very similar to VBScript's MsgBox) containing the value from the text field. The form displays a prompt message (within the `<P></P>` pair) and an input field. The input field has an attribute, `onChange`, that tells JavaScript that when the `Input` field changes it should execute the code given, in this case a call to `myProgram`. The form has a name, `entry` within the `document` context, and the `Input` field has a name, `data` within the entry form context. Thus within the JavaScript program we can refer to the value of the field as:

```
document.entry.data.value
```

I'm not going to show the address book example in JavaScript because the HTML aspects become more complex and the use of functions increases and I want to wait till we have covered those in their own topic.

A Simple Python GUI Tool

Creating a full blown GUI is quite a challenge and we need to cover a lot more material before being able to tackle it. However there is a module called EasyGui which makes it possible to at least interact with the user in a GUI like fashion via dialog boxes. EasyGui is not part of the standard Python library of modules but must be downloaded separately from:

```
http://easygui.sourceforge.net/current_version/index.html
```

You will also find a full tutorial on that site showing all of the different options possible. We will look at three of the simplest:

- A Message box for displaying output,
- An Input box for getting a string from the user
- An Input box for getting a number from the user

The first thing to do is download the package and copy the python file into the site-packages folder within the Lib folder of your Python installation. On my PC that is:

```
C:\Python3\Lib\site-packages
```

Then you can import the module as usual and access the various functions. Here is a simple input/output sequence:

```
import easygui
name = easygui.enterbox("What is your name?", "Name Dialog")
easygui.msgbox("Hello "+name+" nice to meet you!", "Greeting Dialog")
```

Notice the use of the module name to prefix the functions. Also notice that the second string we passed into the function is displayed as the title of the dialog box.

I think you will agree that using EasyGui is nearly as easy to use as `input()`! Let's finally look at our multiplication example with EasyGui:

```
import easygui

output = ""
multiplier = easygui.integerbox("Which mltiplier?", "Multiplier dialog")
for j in range(1,13):
  output = output + "%d x %d = %d\n" % (j, multiplier, j * multiplier)

easygui.msgbox(output, "Multiplication table")
```

There are many other options in EasyGui that will make your programs look much more like a GUI application. Experiment with the functions available and try to repeat some of our other examples, such as the address book using EasyGUI.

Next we look at a different kind of user interaction that is used where the program just needs some initial values and then generates it's output with no further intervention from the user.

A word about *stdin* and *stdout*

NOTE: *stdin* is a bit of computer jargon for the standard input device (usually the keyboard). *stdout* refers to the standard output device (usually the screen). You will quite often see references to the terms stdin and stdout in discussions about programming. (There is a third, less commonly used term, *stderr*, which is where all console error messages are sent. Normally stderr appears in the same place as stdout.) These terms are often called data *streams* since data appears as a stream of bytes flowing to the devices. stdin and stdout are made to look like files (we'll get to those shortly) for consistency with file handling code.

In Python they all live in the `sys` module and are called `sys.stdin` and `sys.stdout`. `input()` uses stdin automatically and `print()` uses stdout. We can also read from stdin and write to stdout directly and this can offer some advantages in terms of fine control of the input and output. Here is an example of reading from stdin:

```
import sys
print( "Type a value: ", end='')  # prevents newline
value = sys.stdin.readline()  # use stdin explicitly
print( value )
```

It is almost identical to:

```
print( input("Type a value: ") )
```

The advantage of the explicit version is that you can do fancy things like make `stdin` point to a real file so the program reads its input from the file rather than the terminal - this can be useful for long testing sessions whereby instead of sitting typing each input as requested we simply let the program read its input from a file. (This has the added advantage of ensuring that we can run the test repeatedly, sure that the input will be exactly the same each time, and so hopefully will the output. This technique of repeating previous tests to ensure that nothing got broken is called *regression testing* by programmers.)

Finally here is an example of direct output to `sys.stdout` that can likewise be redirected to a file. `print` is nearly equivalent to:

```
sys.stdout.write("Hello world\n") # \n= newline
```

Of course we can usually achieve the same effects using format strings if we know what the data looks like but if we don't know what the data will look like till runtime then its often easier to just send it to stdout rather than try to build a complex format string at runtime.

Redirecting stdin & stdout

So how do we redirect stdin and stdout to files? We can do it directly within our program using the normal Python file handling techniques which we will cover shortly, but the easiest way is to do it via the operating system.

This is how the operating system commands work when we use redirection at the command prompt:

```
C:> dir
C:> dir > dir.txt
```

The first command prints a directory listing to the screen. The second prints it to a file. By using the `'>'` sign we tell the program to redirect `stdout` to the file `dir.txt`.

We would do the same with a Python program like this:

```
$ python myprogram.py > result.txt
```

Which would run `myprogram.py` but instead of displaying the output on screen it would write it to the file `result.txt`. We could see the output later using a text editor like notepad.

(Note that the `$` prompt shown above is the standard for Linux users - just in case they were feeling neglected!)

To get stdin to point at a file we simply use a < sign rather than a > sign. Here is a complete example:

First create a file called `echoinput.py` containing the following code:

```
import sys
```

```
inp = sys.stdin.readline()
while inp.strip() != '':
   print( inp )
   inp = sys.stdin.readline()
```

Note: The `strip()` simply chops off the newline character that is retained when reading from stdin, `input()` does that for you as a convenience. You can now try running that from a command prompt:

```
$ python echoinput.py
```

The result should be a program that echos back anything you type until you enter a blank line.

Now create a simple text file called `input.txt` containing some lines of text. Run the last program again, redirecting input from `input.txt`:

```
$ python echoinput.py < input.txt
```

Python echos back what was in the file. But you might recall that we said that `print()` and `input()` actually use stdin and stdout internally? That means we can replace the stdin stuff in `echoinput.py` with `input()` like this:

```
inp = input()
while inp != '':
   print( inp )
   inp = input()
```

Which is much easier in most cases.

By using this technique with multiple different input files we can quickly and easily test our programs for a variety of scenarios (for example bad data values or types) and do so in a repeatable and reliable manner. We can also use this technique to handle large volumes of data from a file while still having the option to input the data manually for small volumes using the same program. Redirecting stdin and stdout is a very useful trick for the programmer, experiment and see what other uses you can find for it.

There is a known bug in Windows XP that breaks input redirection. (If anyone knows whether this has been fixed in Vista please let me know via the email link at the bottom of the page.) If you start your program by just typing in the script name, rather than explicitly typing in `python` *before it, Windows will not display the results on the console! There is a registry hack to fix this on Microsoft's web site, although even the hack isn't quite correct! You need to look under* `HKEY_CURRENT USER` *instead of* `HKEY_LOCAL_MACHINE` *as recommended on the web page. My recommendation is to always explicitly invoke python when dealing with redirected input or output! (Thanks go to Tim Graber for spotting this and to Tim Peters for telling me about the registry hack to fix it.)*

Command Line Parameters

One other type of input is from the command line. For example when you run your text editor from an operating system command line, like:

```
$ EDIT Foo.txt
```

What happens is that the operating system calls the program called EDIT and passes it the name of the file to edit, Foo.txt in this case. So how does the editor read the filename?

In most languages the system provides an array or list of strings containing the command line words. Thus the first element will contain the command itself, the second element will be the first argument, etc. There may also be some kind of magic variable (often called something like *argc*, for "argument count") that holds the number of elements in the list.

In Python that list is held by the `sys` module and called `argv` (for 'argument values'). Python doesn't need an `argc` type value since the usual `len()` method can be used to find the length of the list, and in most cases we don't even need that since we just iterate over the list using Python's `for` loop, like this:

```
import sys
for item in sys.argv:
    print( item )

print( "The first argument was:", sys.argv[1] )
```

Note that this only works if you put it in a file (say args.py) and execute it from the operating system prompt like this:

```
C:\PYTHON\PROJECTS> python args.py 1 23 fred
args.py
1
23
fred
The first argument was: 1
C:\PYTHON\PROJECTS>
```

VBScript and JavaScript

Being web page based the concept of command line arguments doesn't really arise. If we were using them within Microsoft's Windows Script Host environment the situation would be different, and WSH provides a mechanism to extract such arguments from a `WshArguments` object populated by WSH at run time.

That's really as far as we'll go with user input in this course. It's very primitive but you can write useful programs with it. In the early days of Unix or PCs it's the only kind of interaction you got. Of course GUI programs read input too and we will look more closely at how that's done much later in the tutorial.

Points to remember

- Use `input()` for reading strings.
- `input()` can display a string to prompt the user.
- EasyGui provides a GUI style mechanism equivalent to `input()` and `print()`
- Command line arguments can be obtained from the `argv` list imported from the `sys` module in Python, where the first item is the name of the program.

Previous Next Contents

If you have any questions or feedback on this page send me mail at: alan.gauld@btinternet.com

Decisions, Decisions

What will we cover?
• The 3rd programming construct - Branching • Single branches and multiple branches • Using Boolean expressions

The 3rd of our fundamental building blocks is branching or *conditional statements*. These are simply terms to describe the ability of our programs to execute one of several possible sequences of code (branches) depending on some condition.

Back in the early days of Assembler programming the simplest branch was a `JUMP` instruction where the program literally jumped to a specified memory address, usually if the result of the previous instruction was zero. Amazingly complex programs were written with virtually no other form of condition possible - vindicating Dijkstra's statement about the minimum requirements for programming. When high level languages came along a new version of the `JUMP` instruction appeared called `GOTO`. In fact QBASIC, which is still supplied on the CD ROM with older versions of Windows (pre XP), still provides GOTO and, if you have QBASIC installed, you can try it out by typing the following bit of code:

```
10 PRINT "Starting at line 10"
20 J = 5
30 IF J < 10 GOTO 50
40 PRINT "This line is not printed"
50 STOP
```

Notice how even in such a short program it takes a few seconds to figure out what's going to happen. There is no structure to the code, you have to literally figure it out as you read it. In large programs it becomes impossible. For that reason most modern programming languages, including Python, VBScript and JavaScript, either don't have a direct JUMP or GOTO statement or discourage you from using it. So what do we use instead?

The *if* statement

The most intuitively obvious conditional statement is the `if, then, else` construct. It follows the logic of English in that *if* some boolean condition (see below for more about this aspect of things) is true *then* a block of statements is executed, otherwise (or *else*) a different block is executed.

Python

It looks like this in Python:

```
import sys  # only to let us exit
print( "Starting here" )
j = 5
if j > 10:
    print( "This is never printed" )
else:
    sys.exit()
```

Hopefully that is easier to read and understand than the previous `GOTO` example. Of course we can put any test condition we like after the `if`, so long as it evaluates to `True` or `False`, i.e. a boolean value. Try changing the > to a < and see what happens.

VBScript

VBScript looks quite similar:

```
<script type="text/vbscript">
MsgBox "Starting Here"
DIM J
J = 5
If J > 10 Then
    MsgBox "This is never printed"
Else
    MsgBox "End of Program"
End If
</script>
```

It's very nearly identical, isn't it? The main difference is the use of `End If` to indicate the end of the construct.

And JavaScript too

And of course JavaScript has an `if` statement too:

```
<script type="text/javascript">
var j;
j = 5;
if (j > 10){
           document.write("This is never printed");
           }
else {
     document.write("End of program");
     }
</script>
```

Notice that JavaScript uses curly braces to define the blocks of code inside the `if` part and the `else` part. Also the boolean test is contained in parentheses and there is no explicit keyword `then` used. On a point of style, the curly braces can be located anywhere, I have chosen to line them up as shown purely to emphasize the block structure. Also if there is only a single line within the block (as we have here) the braces can be omitted entirely, they are only needed to group lines together into a single block. However, many programmers like to always include the braces since they avoid inconsistencies and avoid having to go back and add them if we add one extra line to a block.

Boolean Expressions

You might remember that in the Raw Materials section we mentioned a *Boolean* type of data. We said it had only two values: `True` or `False`. We very rarely create a Boolean variable but we often create temporary Boolean values using *expressions*. An expression is a combination of variables and values combined by operators to produce a resultant value. In the following example:

```
if x < 5:
  print( x )
```

`x < 5` is the expression and the result will be `True` if x is less than 5 and `False` if x is greater than or equal to 5.

Expressions can be arbitrarily complex provided they evaluate to a single final value. In the case of a branch that value must be either `True` or `False`. However, the definition of these two values varies from language to language. In many languages `False` is the same as `0` or a non-existent value (often called `NULL`, `Nil` or `None`). Thus an empty list or string evaluates to false in a Boolean context. Python works this way and this means we can use a `while` loop to process a list until the list is empty, using something like:

```
while aList:
   # do something here
```

Or we can use an `if` statement to test whether a list is empty without resorting to the `len()` function like this:

```
if aList:
   # do something here
```

Finally we can combine Boolean expressions using Boolean operators which can often cut down the number of `if` statements we need to write.

Consider this example:

```
if value > maximum:
   print( "Value is out of range!" )
else if value < minimum:
   print( "Value is out of range!" )
```

Notice that the block of code executed is identical. We can save some work, both for us and for the computer, by combining both of the tests into a single test like this:

```
if (value < minimum) or (value > maximum):
   print( "Value is out of range!" )
```

Notice we combined both tests using a boolean `or` operator. This is still a single expression because Python evaluates the combined set of tests to a single result. You can think of it as evaluating the first set of parentheses, then the second set of parentheses and finally combines the two calculated values to form the final single value, either True or False. (In practice Python uses a slightly more efficient technique known as *short-circuit evaluation* which we discuss in the Functional Programming topic)

Very often if we think carefully about the tests we need to carry out in our natural language then we will find ourselves using conjunctions like 'and', 'or' and 'not'. If so there's a very good chance we can write a single combined test rather than many separate ones.

Chaining if statements

You can go on to chain these if/then/else statements together by *nesting* them one inside the other. Here is an example in Python:

```
# Assume price created previously...
price = int(input("What price? "))
```

```
if price == 100:
    print( "I'll take it!" )
else:
    if price > 500:
        print( "No way Jose!" )
    else:
        if price > 200:
            print( "How about throwing in a free mouse mat?" )
        else:
            print( "price is an unexpected value!" )
```

Note 1:we used == (that's a double = sign) to test for equality in the first `if` statement, whereas we use = to assign values to variables. Using = when you mean to use == is one of the more common mistakes in programming Python, fortunately Python warns you that it's a syntax error, but you might need to look closely to spot the problem.

Note 2:A subtle point to notice is that we perform the greater-than tests from the highest value down to the lowest. If we did it the other way round the first test, which would be `price > 200` would always be true and we would never progress to the `> 500` test. Similarly if using a sequence of less-than tests you must start at the lowest value and work up. This is another very easy trap to fall into.

VBScript & JavaScript

You can chain `if` statements in VBScript and JavaScript too but as it's pretty self evident I'll only show a VBScript example here:

```
<script type="text/vbscript">
DIM Price
price = InputBox("What's the price?")
price = CInt(price)
If price = 100 Then
   MsgBox "I'll take it!"
Else:
    If price > 500 Then
        MsgBox "No way Jose!"
    Else:
        If price > 200 Then
            MsgBox "How about throwing in a free mouse mat too?"
        Else:
            MsgBox "price is an unexpected value!"
        End If
    End If
End If
</script>
```

The only things to note here are that there is an `End If` statement to match every `If` statement and that we used the VBScript conversion function `CInt` to convert from the input string value to an integer.

Case statements

One snag with chaining, or nesting `if/else` statements is that the indentation causes the code to spread across the page very quickly. A sequence of nested `if/else/if/else...` is such a common construction that many languages provide a special type of branch for it.

This is often referred to as a `Case` or `Switch` statement and the JavaScript version looks like:

```
<script type="text/javascript">
function doArea(){
   var shape, breadth, length, area;
   shape   = document.area.shape.value;
   breadth = parseInt(document.area.breadth.value);
   len     = parseInt(document.area.len.value);
   switch (shape){
       case 'Square':
           area = len * len;
           alert("Area of " + shape + " = " + area);
           break;
       case 'Rectangle':
           area = len * breadth;
           alert("Area of " + shape + " = " + area);
           break;
       case 'Triangle':
           area = len * breadth / 2;
           alert("Area of " + shape + " = " + area);
           break;
       default: alert("No shape matching: " + shape)
       };
}
</script>

<form name="area">
Length:  <input type="text" name="len">
Breadth: <input type="text" name="breadth">
Shape:   <select name="shape" size=1 onChange="doArea()">
           <option value="Square">Square
           <option value="Rectangle">Rectangle
           <option value="Triangle">Triangle
         </select>
</form>
```

The HTML form code just allows us to capture the details and then when the user selects a shape it calls our JavaScript function. The first few lines simply create some local variables and convert the strings to integers where needed. The bold section is the bit we are really interested in. It selects the appropriate action based on the shape value, notice, by the way, that the parentheses around `shape` are required. Each block of code within the case structure is not marked using curly braces, as you might expect, but is instead terminated by a `break` statement. The entire set of `case` statements for the `switch` is, however, bound together as a block by a single set of curly braces.

Finally note the final condition is `default` which is simply a catch-all for anything not caught in the preceding `Case` statements.

Why not see if you can extend the example to cover circles as well? Remember to add a new option to the HTML form as well as a new case to the switch.

VBScript Select Case

VBScript has a version too:

```
<script type="text/vbscript">
Dim shape, length, breadth, SQUARE, RECTANGLE, TRIANGLE
SQUARE = 0
RECTANGLE = 1
TRIANGLE = 2
shape  = CInt(InputBox("Square(0),Rectangle(1) or Triangle(2)?"))
```

```
length = CDbl(InputBox("Length?"))
breadth  = CDbl(InputBox("Breadth?"))
Select Case shape
 Case SQUARE
   area = length * length
   MsgBox "Area = "  & area
 Case RECTANGLE
   area = length * breadth
   MsgBox "Area = " & area
 Case TRIANGLE
   area = length * breadth / 2
   MsgBox "Area = " & area
 Case Else
   MsgBox "Shape not recognized"
End Select
</script>
```

As with the JavaScript example the first few lines simply collect the data from the user and convert it into the right type. The bold `Select` section shows the VBScript case construct with each successive `Case` statement active as a block terminator for the previous one. The whole `Select` construct is closed with the `End Select` statement. Finally there is a `Case Else` clause which, like the `default` in JavaScript catches anything not caught in the `Case`s above.

One other feature worth pointing out is the use of *Symbolic Constants* instead of numbers. That is the uppercase variables `SQUARE, RECTANGLE` and `TRIANGLE` are there simply to make the code easier to read. The uppercase names are simply a convention to indicate that they are constant values rather than conventional variables, but VBScript allows any variable name you like.

Python multi-selection

Python does not provide an explicit case construct but rather compromises by providing an easier `if/elseif/else` format:

```
menu = """
Pick a shape(1-3):
   1) Square
   2) Rectangle
   3) Triangle
"""
shape = int(input(menu))
if shape == 1:
   length = float(input("Length: "))
   print( "Area of square = ", length ** 2 )
elif shape == 2:
   length = float(input("Length: "))
   width = float(input("Width: "))
   print( "Area of rectangle = ", length * width )
elif shape == 3:
   length = float(input("Length: "))
   width = float(input("Width: "))
   print( "Area of triangle = ", length * width/2 )
else:
   print( "Not a valid shape, try again" )
```

Note the use of `elif` and the fact that the indentation (all important in Python) does not change (unlike the nested if statement example). It's also worth pointing out that *both* this technique and the earlier nested if/else example are equally valid, the `elif` technique is just a little easier to read if there are many tests. The final condition is an `else` which catches anything not caught by the previous tests, just like the `default` in JavaScript and `Case Else` in VBScript.

VBScript also provides a slightly more cumbersome version of this technique with `ElseIf...Then` which is used in exactly the same way as the Python `elif` but is rarely seen since `Select Case` is easier to use.

Putting it all together

So far many of our examples have been pretty abstract. To conclude let's take a look at an example that uses nearly everything we've learned about so far to introduce a common programming technique, namely displaying menus for controlling user input.

Here is the code, followed by a brief discussion:

```
menu = """
Pick a shape(1-3):
   1) Square
   2) Rectangle
   3) Triangle

   4) Quit
"""
shape = int(input(menu))
while shape != 4:
   if shape == 1:
      length = float(input("Length: "))
      print( "Area of square = ", length ** 2 )
   elif shape == 2:
      length = float(input("Length: "))
      width = float(input("Width: "))
      print( "Area of rectangle = ", length * width )
   elif shape == 3:
      length = float(input("Length: "))
      width = float(input("Width: "))
      print( "Area of triangle = ", length * width / 2 )
   else:
      print( "Not a valid shape, try again" )
   shape = int(input(menu))
```

We've added just three lines (in bold) to the previous Python example but in so doing have significantly enhanced the usability of our program. By adding a 'Quit' option to the menu, plus a while loop we have provided the capability for the user to keep on calculating sizes of different shapes until she has all the information she needs. There is no need to rerun the program manually each time. The only other line we added was to repeat the `input(menu)` shape selection so that the user gets the chance to change the shape and, ultimately, to quit.

What the program does is create the illusion to the user that the program knows what they want to do and does it correctly, acting differently depending on what they input. In essence the user appears to be in control, whereas in fact, the programmer is in control since he has anticipated all the valid inputs and how the program will react. The intelligence on display is that of the programmer, not the machine - computers after all are stupid!

You see how easily we can extend our program just by adding a few lines and combining sequences (the blocks that calculate the area), loops (the while loop) and conditionals (the if/elif structure). Dijkstra's three building blocks of programming. Having covered all three you can, in theory, now go out and program anything, but there are a few more techniques we can learn to make things a bit easier, so don't rush off just yet.

Conditional Expressions

One form of branching that is very common is where we want to assign a different value to a variable depending on some condition. This is very easily done using a standard `if/else` condition, like so:

```
if someCondition:
   value = 'foo'
else:
   value = 'bar'
```

However this is so common that some languages provide a shortcut that is called a *conditional expression* structure. In Python this looks like:

```
value = 'foo' if someCondition else 'bar'
```

This is identical to the longer form above. VBScript doesn't have such a structure but JavaScript does provide something similar using a slightly cryptic syntax:

```
<script type="text/javascript">
var someCondition = true;
var s;

s = (someCondition ? "foo" : "bar");
document.write(s);
</script>
```

Notice the strange syntax in the parentheses? Basically it does the same as the Python version but just uses a more concise set of symbols. Basically it says if the expression before the question mark is true then return the value following the question mark, otherwise return the value after the colon. Notice also that I used parentheses in this example. These aren't required but they often make it more obvious what is going on and I recommend them when using conditional expressions, even in Python.

These kinds of stylistic shortcuts can be convenient but many programmers find them a but clumsy and prefer not to use them. My personal advice is to use them where it makes sense, usually in simple cases, but avoid them if it starts to make the code look overly complex.

Modifying collections from inside loops

We mentioned in the looping topic that modifying a collection from inside a loop was a difficult thing to do, but never got round to explaining how to do it! The reason is, we had to wait for branching to be explained first. So here is the solution:

If we need to modify the elements of a collection in place we can use a `while` loop to make the changes as we iterate over it. We can do this because in a `while` construct we have explicit control over the index, unlike the situation in a `for` loop where the index is automatically updated. Let's see how to delete all zeros from a list:

```
myList = [1,2,3,0,4,5,0]
index = 0
while index < len(myList):
   if myList[index] == 0:
      del(myList[index])
   else:
      index += 1
print( myList )
```

The thing to note here is that we do **not** increment the index if we remove an item, we rely on the deletion moving everything up so that the old index value now points at the next item in the collection. We use an `if/else` branch to control when we increment the index. It's very easy to make a mistake doing this kind of thing so test your code carefully. There is another set of Python functions which are specifically designed for manipulating list contents and we look at them in the Functional Programming topic in the advanced section of the tutorial.

Things to Remember

- Use `if/else` to branch
- The `else` is optional
- Multiple decisions can be represented using a `Case` or `if/elif` construct
- Boolean expressions return `True` or `False`
- Combining menus with `Case` constructs allows us to build a wide range of user controlled applications.

Previous Next Contents

If you have any questions or feedback on this page send me mail at: alan.gauld@btinternet.com

Programming with Modules

What will we cover?
• What modules are about • Functions as modules • Using module files • Writing our own functions and modules • An introduction to Windows Script Host

What's a Module?

The 4th element of programming involves the use of *modules*. In fact it's not strictly necessary, and using what we've covered so far you can actually write some pretty impressive programs. However, as the programs get bigger it becomes harder and harder to keep track of what's happening and where. We really need a way to abstract away some of the details so that we can think about the problems we are trying to solve rather than the minutiae of how the computer works. To some extent that's what Python, VBScript and JavaScript already do for us with their built in capabilities - they prevent us from having to deal with the hardware of the computer, how to read the individual keys on the keyboard etc.

The idea of programming with modules is to allow the programmer to *extend* the built in capabilities of the language. It packages up bits of program into modules that we can 'plug in' to our programs. The first form of module was the *subroutine* which was a block of code that you could jump to (rather like the `GOTO` mentioned in the branching section) but when the block completed, it could jump back to wherever it was *called* from. That particular style of modularity is known as a *procedure* or *function*. In Python and some other languages the word *module* has taken on a more specific meaning which we will look at shortly, but first let's consider functions a bit more closely.

Using Functions

Before considering how to create functions let's look at how we use the many, many, functions that come with any programming language (the set of standard functions available for any given language often called that language's *standard library*).

We've already seen some functions in use and listed others in the operators section of the Raw Materials topic. Now we'll consider what these have in common and how we can use them in our programs.

The basic structure of a *function call* is as follows:

```
aValue = someFunction ( anArgument, another, etc... )
```

That is, the variable `aValue` takes on the value obtained by calling a function called `someFunction`. The function can accept, inside parentheses, zero or many *arguments* which it treats like internal variables. Functions can call other functions internally. In most programming languages (although not all), even if there are no arguments, we must still provide the parentheses when calling a function.

Let's consider some examples in our various languages to see how this works:

VBScript: Mid(aString, start, length)

This returns the next `length` characters starting at the `start` in `aString`.

```
<script type="text/vbscript">
Dim time
time = "MORNING EVENING AFTERNOON"
MsgBox "Good" & Mid(time, 8, 8)
</script>
```

This displays "Good EVENING". One feature to note about VBScript is that it does not require parentheses to group the function's arguments, spaces are usually sufficient, as we have been doing with `MsgBox`. However if we combine two functions (as we do here) then the inner one must use parentheses, my advice is: if in doubt, use the parentheses.

VBScript: Date

This returns the current system date.

```
<script type="text/vbscript">
MsgBox Date
</script>
```

There's not much more I can say about that, except that there's a whole bunch of other date functions for extracting the day, week, hour etc.

JavaScript: startString.replace(searchString, newString)

Returns a new string with the `searchString` replaced by `newString,` in `startString`

```
<script type="text/javascript">
var r,s = "A long and winding road";
document.write("Original = " + s + "<BR>");
r = s.replace("long", "short");
document.write("Result  = " + r);
</script>
```

Note: almost everything in JavaScript is an example of a special type of function called a *method*. A method is a function that is associated with an *object* (as discussed in the Raw Materials topic and in much more detail later in the tutorial). The main thing to note here is that the function is "attached" to the string `s` by the dot operator which means that `s` is the string that we will be performing the substitution upon.

This is nothing new. We have been using the `write()` method of the `document` object to display the output from our JavaScript programs (using `document.write()`) since the beginning of the tutorial, I just haven't explained the reason behind the dual name format up until now.

Back in the Raw Materials topic we mentioned that JavaScript used a module called `Math` to do exponentiation. We can see that now.

JavaScript: Math.pow(x,y)

The function in the Math module that we use is `pow(x,y)`, which raises x to the power y:

```
<script type="text/javascript">
```

```
document.write( Math.pow(2,3) );
</script>
```

Python: pow(x,y)

Python also has a `pow()` function which raises `x` to the power `y`.

```
>>> x = 2   #  we'll use 2 as our base number
>>> for y in range(0,11):
...    print( pow(x,y) )    # raise 2 to power y, i.e. 0-10
```

Here we generate values of `y` from 0 to 10 and call the built-in `pow()` function passing 2 arguments: `x` and `y`. On each iteration of the loop the current values of `x` and `y` are substituted into the `pow()` call and the result is printed.

Note: The Python exponentiation operator, `**` is equivalent to the `pow()` function.

Python: dir(m)

Another useful function built in to python is `dir` which, when passed the name of a module displays all of the exported names within the module - including all of the variables and functions that you can use. Python comes with lots of modules, although we haven't really discussed them up till now. The `dir` function gives back a list of valid names - often functions - in that module. Try it on the built-in functions:

```
>>> print dir(__builtins__)
```

Note 1: `builtins` is one of Python's "magic" words so once again we need to surround it with double underscores - that's two underscores at each end.

Note 2: To use `dir()` on any other module you need to `import` the module first otherwise Python will complain that it doesn't recognize the name.

```
>>> import sys
>>> dir(sys)
```

You will recall that we met the `sys` module away back in our first sequences topic. In the output from that last `dir` you should spot our old friend `exit` buried in the middle of all the other stuff in sys.

Before doing much else we'd better talk about Python modules in a bit more detail.

Using Modules

Python is an extremely extensible language in that you can add new capabilities by `importing` modules. We'll see how to create modules shortly but for now we'll play with some of the standard modules that ship with Python.

sys

We met `sys` already when we used it to `exit` from Python. It has a whole bunch of other useful functions too, as we saw with the `dir` function above. To gain access to these we must `import sys`:

```
import sys # make functions available
print( sys.path ) # show where Python looks for modules
sys.exit() # prefix with 'sys'
```

If we know that we will be using the functions a lot and that they won't have the same names as functions we have already imported or created then we can do:

```
from sys import *  # import all names in sys
print( path ) # can use without specifying prefix 'sys'
exit()
```

The big danger with this approach is that two modules could define functions with the same name and then we could only use the second one that we import (because it will override the first). If we only want to use a couple of items then it's safer to do it this way:

```
from sys import path, exit  # import the ones we need
exit() # use without specifying prefix 'sys'
```

Note that the names we specify do not have the parentheses following them. If that was the case we would attempt to execute the functions rather than import them. The name of the function is all that is given.

Finally I'd like to show you a shorthand trick that saves some typing. If you have a module with a very long name we can rename the module when we import it. Here is an example:

```
import SimpleXMLRPCServer as s
s.SimpleXMLRPCRequestHandler()
```

Notice that we told Python to consider `s` to be a shorthand for `SimpleXMLRPCServer`. Then to use the functions of the module we only need to type `s.` which is much shorter!

Other Python modules and what they contain

You can import and use any of Python's modules in this way and that includes modules you create yourself. We'll see how to do that in a moment. First though, I'll give you a quick tour of some of Python's standard modules and some of what they offer:

Module name	Description
`sys`	Allows interaction with the Python system: • exit() - exit! • argv - access command line arguments • path - access the system module search path • ps1 - change the '>>>' Python prompt!
`os`	Allows interaction with the operating system: • name - the current operating system, useful for portable programs • system - execute a system command • mkdir - create a directory • getcwd - find the *current working directory*

`re`	Allows manipulation of strings with Unix style regular expressions • search - find pattern anywhere in string • match - find at beginning only • findall - find all occurences in a string • split - break into fields separated by pattern • sub,subn - string substitution
`math`	Allows access to many mathematical functions: • sin,cos etc - trigonometrical functions • log,log10 - natural and decimal logarithms • ceil,floor - ceiling and floor • pi, e - natural constants
`time`	time(and date) functions • time - get the current time (expressed in seconds) • gmtime - convert time in secs to UTC (GMT) • localtime - convert to local time instead • mktime - inverse of localtime • strftime - format a time string, e.g. YYYYMMDD or DDMMYYY • sleep - pause program for *n* seconds
`random`	random number generators - useful for games programming! • randint - generate random integer between inclusive end points • sample - generate random sublist from a bigger list • seed - reset the number generator key

These are just the tip of the iceberg. There are literally dozens of modules provided with Python, and as many again that you can download. (A couple of good sources are the Vaults of Parnassus and, especially for more recent things, the *Cheese shop*.) SourceForge is also home to many Python projects that have useful modules available. Google search is your friend, just include 'python' in the search string. Don;t forget to read the documentation to find out how to do Internet programming, graphics, build databases etc. (I touch on some of these topics in the Applicatons section of this tutorial.)

The important thing to realize is that most programming languages have these basic functions either built in or as part of their standard library. Always check the documentation before writing a function - it may already be there! Which leads us nicely into...

Defining our own functions

OK, so we know how to use the existing functions and modules, but how do we create a new function? Simply by *defining* it. That is we write a statement which tells the interpreter that we are defining a block of code that it should execute, on demand, elsewhere in our program.

VBScript first

So let's create a function that can print out a multiplication table for us for any value that we provide as an argument. In VBScript it looks like:

```
<script type="text/vbscript">
Sub Times(N)
```

```
Dim I
For I = 1 To 12
    MsgBox I & " x " & N & " = " & I * N
Next
End Sub
</script>
```

We start with the keyword `Sub` (for Subroutine) and end the definition with `End Sub`, following the normal VBScript block marker style. We provide a list of *parameters* enclosed in parentheses. The code inside the defined block is just normal VBScript code with the exception that it treats the parameters as if they were local variables. So in the example above the function is called `Times` and it takes a single parameter called `N`. It also defines a local variable `I`. It then executes a loop to display the N times table, using both `N` and `I` as variables.

We can now *call* the new function like this:

```
<script type="text/vbscript">
MsgBox "Here is the 7 times table..."
Times 7
</script>
```

Note 1: We defined a *parameter* called `N` and passed an *argument* of `7`. The parameter (or local variable) `N` inside the function took the value 7 when we called it. We can define as many parameters as we want in the function definition and the calling programs must provide values for each parameter. Some programming languages allow you to define *default values* for a parameter so that if no value is provided the function assumes the default. We'll see this in Python later.

Note 2: We enclosed the parameter, `N`, in parentheses during function definition but, as is usual in VBScript, we did not need to use parentheses when calling the function.

This function does not return a value and is really what is called a procedure, which is, quite simply, a function that doesn't return a value! VBScript differentiates between functions and procedures by having a different name for their definitions. Let's look at a true VBScript function that returns the multiplication table as a single, long string:

```
<script type="text/vbscript">
Function TimesTable (N)
  Dim I, S
  S = N & " times table" & vbNewLine
  For I = 1 to 12
    S = S & I & " x " & N & " = " & I*N & vbNewLine
  Next
  TimesTable = S
End Function

Dim Multiplier
Multiplier = InputBox("Which table would you like?")
MsgBox TimesTable (Multiplier)
</script>
```

It's very nearly identical to the `Sub` syntax, except we use the word `Function` instead of `Sub`. However, notice that you must assign the result to the function name inside the definition. The function returns as a result whatever value the function name contains when it exits:

```
  ...
```

```
  TimesTable = S
End Function
```

If you don't assign an explicit value the function will return a default value, usually zero or an empty string.

Notice also that we had to put parentheses around the argument in the `MsgBox` line. That's because `MsgBox` wouldn't otherwise have been able to work out whether `Multiplier` was to be printed or passed to its first argument which was `TimesTable`. By putting it in parentheses it is clear to the interpreter that the value is an argument of `TimesTable` rather than of `MsgBox`.

Python too

In Python the Times function looks like:

```
def times(n):
    for i in range(1,13):
        print( "%d x %d = %d" % (i, n, i*n) )
```

And is called like:

```
print( "Here is the 9 times table..." )
times(9)
```

Note that in Python procedures are not distinguished from functions and the same name `def` is used to define both. The only difference is that a function which returns a value uses a `return` statement, like this:

```
def timesTable(n):
   s = ""
   for i in range(1,13):
       s = s + "%d x %d = %d\n" % (i,n,n*i)
   return s
```

As you see it's very simple, just return the result using a `return` statement. (If you don't have an explicit `return` statement Python automatically returns a default value called `None` which we usually just ignore.)

We can then simply print the result of the function like so:

```
print( timesTable(7) )
```

Although we haven't followed this advice throughout this tutorial, it is usually best to avoid putting `print` statements inside functions. Instead, get them to `return` the result and print that from outside the function. That makes your functions much more reusable, in a wider variety of situations.

A Note About Parameters

Sometimes beginners find it hard to understand the role of parameters in function definitions. That is, whether they should define a function like this:

```
def f(x): # can use x within the function...
```

or

```
x = 42
def f(): # can use x within the function...
```

The first example defines a parameter x and uses it inside the function, whereas the second directly uses a variable defined outside the function. Since the second method (usually) works why bother defining the parameter?

We have already said that the parameters act as local variables, that is, ones which are only usable inside the function. And we've said that the user of the function can pass in arguments to those parameters. So the parameter list acts like a gateway for data moving between the main program and the inside of the function.

The function can see some data outside the function (see the What's in a Name? topic for more on that). However if we want the function to have maximum re-usability across many programs we want to minimise its dependence on external data. Ideally all the data that a function needs to work properly should be passed into it via its parameters.

If the function is defined inside a module file it is permissible to read data defined in that same module, but even that will reduce the flexibility of your function. Of course if a lot of data is involved it may mean that you need a high number of parameters but we can reduce that to a manageable level by using data collections: lists, tuples and dictionaries etc. Also, in Python and some other languages, we can reduce the number of actual parameter values we need to provide by using something called *default arguments* which we discuss in the next section.

Default Values

You might recall that I mentioned the use of default values earlier? This refers to a way of defining function parameters that, if not passed as arguments explicitly, will take on a default value. One sensible use for these would be in a function which returns the day of the week. If we call it with no value we mean today, otherwise we provide a day number as an argument. Something like this:

```
import time

# a day value of None => today
def dayOfWeek(DayNum = None):
    # match day order to Python's return values
    days = ['Monday','Tuesday',
            'Wednesday','Thursday',
            'Friday', 'Saturday', 'Sunday']

    # check for the default value
    if DayNum == None:
        theTime = time.localtime(time.time())
        DayNum = theTime[6] # extract the day value
    return days[DayNum]
```

Note: We only need to use the time module if the default parameter value is involved, therefore we could defer the import operation until we need it. This would provide a slight performance improvement if we never had to use the default value feature of the function, but it is so small, and breaks the convention of importing at the top, that the gain isn't worth the extra confusion.

Now we can call this with:

```
print( "Today is: %s" % dayOfWeek() )
# remember that in computer speak we start from 0
# and in this case we assume the first day is Monday.
print( "The third day is %s" % dayOfWeek(2) )
```

Counting Words

Another example of a function which returns a value might be one which counts the words in a string. You could use that to calculate the words in a file by adding the totals for each line together.

The code for that might look something like this:

```
def numwords(s):
    s = s.strip() # remove "excess" characters
    list = s.split() # list with each element a word
    return len(list) # number of elements in list is the number of words in s
```

That defines the function, making use of some of the built-in string methods which we mentioned in passing in the Raw Materials chapter.

We would use it by doing something like this:

```
for line in file:
    total = total + numwords(line) # accumulate totals for each line
print( "File had %d words" % total )
```

Now if you tried typing that in, you'll know that it didn't work. Sorry! What I've done is a common design technique which is to sketch out how I think the code should look but not bothered to use the absolutely correct code. This is sometimes known as *Pseudo Code* or in a slightly more formal style *Program Description Language (PDL)*.

One other thing that this illustrates is why it is better to return a value from a function and print the result outside the function rather than to print the value inside the function. If our function had printed the length rather than returning it we could not have used it to count the total words in the file, we would simply have gotten a long list of the length of each line. By returning the value we can choose to use the value that way or, as we did here, simply store it in a variable for further processing - in this case taking the total count. It is a very important design point to separate the display of your data (via print) from the processing of the data (in the function). A further advantage is that if we print the output it will only be useful in a command line environment, but if we return the value we can display it in a web page or a GUI too. Separating processing from presentation is very powerful, try to always return values from functions rather than printing them. The exception to this rule is where you create a function specifically to print out some data, in which case try to make this obvious by using the word print or display in the function name.

Once we've had a closer look at file and string handling, a little later in the course, we'll come back to this example and write it for real.

JavaScript Functions

We can also create functions in JavaScript, of course, and we do so using the `function` command, like so:

```
<script type="text/javascript">
```

```
var i, values;

function times(m) {
    var results = new Array();
    for (i = 1; i <= 12; i++) {
        results[i] = i * m;
        }
    return results;
}


// Use the function
values = times(8);

for (i=1;i<=12;i++){
    document.write(values[i] + "<BR>");
    }
</script>
```

In this case the function doesn't help much, but hopefully you can see that the basic structure is very similar to the Python and VBScript function definitions. We'll see more complex JavaScript functions as we go through the tutor. In particular JavaScript uses functions to define objects as well as functions, which sounds confusing, and indeed can be!

Before we move on though, now is a good time to look back at the JavaScript example in Talking to the User, where we used JavaScript to read input from a web form. The code looked like this:

```
<script type="text/javascript">
function myProgram(){
   alert("We got a value of " + document.entry.data.value);
}
</script>

<form name='entry'>
<P>Type value then click outside the field with your mouse</P>
<Input Type='text' Name='data' onChange='myProgram()'>
</form>
```

Looking at that we can now see that what we did was define a JavaScript function called `myProgram` and then tell the form to call that function when the `Input` field changed. We'll explain this further in the topic on Event Driven programming

A Word of Caution

Functions are very powerful because they allow us to extend the language, they also give us the power to *change* the language by defining a new meaning for an existing function (some languages don't allow you to do this), but this is usually a bad idea unless carefully controlled (we'll see a way to control it in a minute). By changing the behavior of a standard language function your code can become very difficult for other people (or even you, later on) to read, since they expect the function to do one thing but you have redefined it to do another. Thus it is good practice not to change the basic behavior of built in functions.

One way to get round this limitation of not changing built in behavior but still using a meaningful name for our functions is to put the functions inside either an object or a module which provides its own local context. We'll look at the object approach in the OOP topic a little later but for now let's see how we go about creating our own modules.

Creating our own modules

So far we have seen how to create our own functions and call these from other parts of our program. That's good because it can save us a lot of typing and, more importantly, makes our programs easier to understand because we can forget about some of the details after we create the function that hides them. (This principle of wrapping up the complex bits of a program inside functions is called *information hiding* for fairly obvious reasons.) But how can we use these functions in other programs? The answer is that we create a *module*.

Python Modules

A module in Python is nothing special. It's just a plain text file full of Python program statements. Usually these statements are function definitions. Thus when we type:

```
import sys
```

we tell the Python interpreter to read that module, executing the code contained in it and making the names that it generated available to us in our file. It is almost like making a copy the contents of `sys.py` into our program, like a cut n' paste operation. (it's not really like that but the concept is similar. In fact `sys` is a special kind of module that doesn't actually have a `sys.py` file, but we will ignore that for now!). In fact in some programming languages (noteably C and C++) the translator literally does sometimes copy module files into the current program as required.

So to recap, we create a module by creating a Python file containing the functions we want to reuse in other programs. Then we just import our module exactly like we do the standard modules. Easy eh? Let's do it.

Copy the function below into a file by itself and save the file with the name `timestab.py`. You can do this using IDLE or Notepad or any other editor that saves plain text files. Do not use a Word Processing program since they tend to insert all sorts of fancy formatting codes that Python will not understand.

```
def print_table(multiplier):
    print( "--- Printing the %d times table ---" % multiplier )
    for n in range(1,13):
        print( "%d x %d = %d" % (n, multiplier, n*multiplier) )
```

Now at the Python prompt type:

```
>>> import timestab
>>> timestab.print_table(12)
```

Heh presto! You've created a module, imported it and used the function defined inside it.

Important Note: If you didn't start Python from the same directory that you stored the `timestab.py` file then Python might not have been able to find the file and reported an error. If so then you can create an environment variable called PYTHONPATH that holds a list of valid directories to search for modules (in addition to the standard modules supplied with Python). I find it convenient to define a folder in my PYTHONPATH and store all my reusable module files in that folders. Obviously you should test your modules thoroughly before moving them into that folder.

Creating environment variables is a platform specific operation which I assume you either know how to do or can find out! For example Windows XP users can use the Start->Help & Support facility to search for Environment Variables and see how to create them.

Modules in VBScript and JavaScript

What about VBScript? That's more complex.... In VBScript itself and other older varieties there is no real module concept. Instead, VBScript relies on the creation of objects to reuse code between projects. We look at this later in the tutorial. Meantime you will have to manually cut n' paste from previous projects into your current one using your text editor.

Note: VBScript's big brother Visual Basic does have a module concept and you can load a module via the *Integrated Development Environment (IDE)* `File|Open Module...` menu. There are a few restrictions as to what kind of things you can do inside a VB module but since we're not using Visual Basic on this course I won't go into that any further. Microsoft make a free version of the latest VB Express version available although you have to register with them before you can use it. If you feel like experimenting this page has more details.

Like VBScript, JavaScript does not offer any direct mechanism for reuse of code files as modules. However there are some exceptions to these in specialised environments such as where JavaScript is used outside of a web page (See the *Windows Script Host* box below for an example).

Windows Script Host

So far we have looked at VBScript and JavaScript as languages for programming within a web browser. That imposes some restrictions including the lack of a way to include a module of reusable code. There is another way to use VBScript (and JavaScript) within a Windows environment, namely *Windows Script Host* or *WSH*. WSH is Microsoft's technology to enable users to program their PCs in the same way that DOS programmers used Batch files. WSH provides mechanisms for reading files and the registry, accessing networked PCs and Printers etc.

In addition WSH v2 includes the ability to include another WSH file and thus provides reusable modules. It works like this, first create a module file called `SomeModule.vbs` containing:

```
Function SubtractTwo(N)
   SubtractTwo = N - 2
End function
```

Now create a WSH script file called, say, `testModule.wsf`, like this:

```
<?xml version="1.0" encoding="UTF-8"?>

<job>
  <script type="text/vbscript" src="SomeModule.vbs" />
  <script type="text/vbscript">
      Dim value, result
      WScript.Echo "Type a number"
      value = WScript.StdIn.ReadLine
      result = SubtractTwo(CInt(value))

      WScript.Echo "The result was " &amp; CStr(result)
  </script>
```

```
</job>
```

You can run it under Windows by starting a DOS session and typing:

```
C:\> cscript testModule.wsf
```

The structure of the .wsf file is XML and the program lives inside a pair of `<job></job>` tags, rather like our `<HTML></HTML>` tags. Inside the first script tag references a module file called `SomeModule.vbs` and the second script tag contains our program which accesses `SubtractTwo` within the `SomeModule.vbs` file. The `.vbs` file just contains regular VBScript code with no XML or HTML tags whatsoever.

Notice that to concatenate the strings for the `WScript.Echo` statement we have to *escape* the ampersand (with `&amp;`) because the statement is part of an XML file! Notice too, that we use the `WScript.Stdin` to read user input, you might recall the sidebar in the User Input topic that discussed stdin and stdout?

This technique works with JavaScript too, or more correctly with Microsoft's version of JavaScript called *JScript*, simply by changing the `type=` attribute. In fact you can even mix languages in WSH by importing a module written in JavaScript and using it in VBScript code, or vice-versa! To prove the point, here is the equivalent WSH script using JavaScript to access the VBScript module:

```
<?xml version="1.0" encoding="UTF-8"?>

<job>
  <script type="text/vbscript" src="SomeModule.vbs" />
  <script type="text/javascript">
      var value, result;
      WScript.Echo("Type a number");
      value = WScript.StdIn.ReadLine();
      result = SubtractTwo(parseInt(value));

      WScript.Echo("The result was " + result);
  </script>
</job>
```

You can see how closely related the two versions are, most of the clever stuff is actually done through the WScript objects and apart from a few extra parentheses the scripts are very much alike.

I won't use WSH very often in this tutor but occasionally we will delve into it when it offers capabilities that I cannot demonstrate using the more restricted web browser environment. For example the next topic will use WSH to show how we can manipulate files using VBScript and JavaScript. There are a few books available on WSH if you are interested, and Microsoft have a large section of their web site dedicated to it, complete with sample programs and development tools etc. You'll find it here: http://msdn.microsoft.com/scripting/

Next we'll take a look at files and text handling and then, as promised, revisit the business of counting words in a file. In fact we're eventually going to create a module of text handling functions for our convenience.

Things to remember
• Functions are a form of module • Functions return values, procedures don't • Python modules normally consist of function definitions in a file • Create new functions with the `def` keyword in Python • Use `Sub` or `Function` in VBScript and `function` in JavaScript

Previous Contents Next

If you have any questions or feedback on this page send me mail at: alan.gauld@btinternet.com

Handling Files

What will we cover?
• How to open a file • How to read and write to an open file • How to close a file. • Building an address book • Handling binary data files • Random Access to file data

Handling files often poses problems for beginners although the reason for this puzzles me slightly. Files in a programming sense are really not very different from files that you use in a word processor or other application: you *open* them, do some work and then *close* them again.

The biggest differences are that in a program you access the file *sequentially*, that is, you read one line at a time starting at the beginning. In practice the word processor often does the same, it just holds the entire file in memory while you work on it and then writes it all back out when you close it. The other difference is that, when programming, you normally open the file as *read only* or *write only*. You can write by creating a new file from scratch (or overwriting an existing one) or by *appending* to an existing one.

One other thing you can do while processing a file is that you can go back to the beginning.

Files - Input and Output

Let's see that in practice. We will assume that a file exists called `menu.txt` and that it holds a list of meals:

```
spam & eggs
spam & chips
spam & spam
```

Now we will write a program to read the file and display the output - like the 'cat' command in Unix or the 'type' command in DOS.

```
# First open the file to read(r)
inp = open("menu.txt","r")
# read the file line by line
for line in inp:
    print( line )
# Now close it again
inp.close()
```

Note 1: `open()` takes two arguments. The first is the filename (which may be passed as a variable or a literal string, as we did here). The second is the *mode*. The mode determines whether we are opening the file for reading(r) or writing(w), and also whether it's for text or binary usage - by adding a 'b' to the 'r' or 'w', as in: `open(fn,"rb")`

Note 2: We read and close the file using functions preceded by the file variable. This notation is known as *method invocation* and is another example of *Object Orientation*. Don't worry about it for now, except to realize that it's related in some ways to modules. You can, if it helps, think of a file variable as being a reference to a module containing functions that operate on files and which we automatically import every time we create a file type variable.

Note 3: We close the file at the end with the `close()` method. In Python, files are automatically closed at the end of the program but it is good practice to get into the habit of closing your files explicitly. Why? Well, the operating system may not write the data out to the file until it is closed (this can boost performance). What this means is that if the program exits unexpectedly there is a danger that your precious data may not have been written to the file! So the moral is: once you finish writing to a file, close it.

Note 4: We have not specified the full path to the file in the code above so the file will be treated as being in the current folder. However we can pass a full path name to `open()` instead of just the file name. There is a wrinkle when using Windows however, because the \ character used to separate folders in a Windows path has a special meaning inside a Python string. So, when specifying paths in Python it is best to always use the / character instead and that
will work on any Operating System including Windows.

Now, consider how you could cope with long files. You couldn't display all of the file on a single screen so we need to pause after each screenful of text. You might use a `line_count` variable which is incremented for each line and then tested to see whether it is equal to 25 (for a 25 line screen). If so, you request the user to press a key (enter, say) before resetting `line_count` to zero and continuing. You might like to try that as an exercise...

Another way of reading a file is to use a while loop and a method of the file object called `readline()`. The advantage of this is that we can stop processing the file as soon as we find the data we need, this can greatly speed things up if processing long files. However it is a little bit more complex, so lets look at the previous example using a while loop:

```
# First open the file to read(r)
inp = open("menu.txt","r")
# read the file and print each line
line = inp.readline()
while line:
    print( line )
    line = inp.readline()
# Now close it again
inp.close()
```

Note: we read the first line before entering the loop so that the test condition would pass. Thereafter we printed each line, read the next and so long as it was not empty went around again. Finally, after exiting the while loop we closed the file. If we wanted to stop at a certain point in the file we would have introduced a branch condition inside the while loop and if it detected the stop condition we simply set the line value to an empty string (`''`) so that the loop will terminate (Recall that an empty value is treated as a `False` boolean value in Python tests).

Really that's all there is to it. You open the file, read it in and manipulate it any way you want to. When you're finished you close the file. However there is one little niggle you may have noticed in the previous example: the lines read from the file have a *newline* character at the end, so you wind up with blank lines using `print()` (which adds its own newline). To avoid that Python provides a string

method called `strip()` which will remove *whitespace*, or non-printable characters, from both ends of a string. (It has cousins which can strip one end only called `rstrip` and `lstrip` too) If we substitute the print() line above with:

```
    print( line.rstrip() )  #only strip right hand end
```

Everything should now work just fine.

To create a 'copy' command in Python, we simply open a new file in write mode and write the lines to that file instead of printing them. Like this:

```
# Create the equivalent of: COPY MENU.TXT MENU.BAK

# First open the files to read(r) and write(w)
inp = open("menu.txt","r")
outp = open("menu.bak","w")

# read file, copying each line to new file
for line in inp:
    outp.write(line)

print( "1 file copied..." )

# Now close the files
inp.close()
outp.close()
```

Did you notice that I added a `print()` statement at the end, just to reassure the user that something actually happened? This kind of *user feedback* is usually a good idea.

Because we wrote out the same `line` that we read in there was no problems with newline characters here. But if we had been writing out strings which we created, or which we had `strip`ped earlier we would have needed to add a newline to the end of the output string, like this:

```
outp.write(line + '\n') # \n is a newline
```

Let's look at how we might incorporate that into our copy program. Instead of simply copying the menu we will add todays date to the top. That way we can easily generate a daily menu from the easily modified text file of meals. All we need to do is write out a couple of lines at the top of the new file before copying the menu.txt file, like this:

```
import time
# Create daily menu based on MENU.TXT
# First open the files to read(r) and write(w)
inp = open("menu.txt","r")
outp = open("menu.prn","w")

# Create todays date string
today = time.localtime(time.time())
theDate = time.strftime("%A %B %d", today)

# Add Banner text and a blank line
outp.write("Menu for %s\n\n" % theDate)

# copy each line of menu.txt to new file
```

```
for line in inp:
    outp.write(line)

print( "Menu created for %s..." % theDate )

# Now close the files
inp.close()
outp.close()
```

Note that we use the `time` module to get todays date (`time.time()`) and convert it into a tuple of values (`time.localtime()`) which are then used by `time.strftime()` (check the documentation for time.strftime to see what else it can do) to produce a string which, when inserted into a title message using string formatting, looks like:

```
Menu for Sunday September 19

Spam & Eggs
Spam &...
```

Although we added two '\n' characters at the end there is only one blank line printed, that's because one of them is the newline at the end of the title itself. Managing the creation and removal of newline characters is one of the more irritating aspects of handling text files.

Some Operating Systems Gotchas

Operating systems handle files in different ways. This introduces some niggles into our programs if we want them to work on multiple operating systems. There are two niggles in particular which can catch people out and we'll look at them here:

Newlines

The whole subject of newlines and text files is a murky area of non standard implementation by different operating systems. These differences have their roots in the early days of data communications and the control of mechanical teleprinters. Basically there are 3 different ways to indicate a new line:

1. A Carriage Return (CR) character ('\r')
2. A Line Feed (LF) character ('\n')
3. A CR/LF pair ('\r\n').

All three techniques are used in different operating systems. MS DOS (and therefore Windows) uses method *3*. Unix (including Linux) uses method *2*. Apple in its original MacOS used method *1*, but now uses method *2* since MacOS X is really a variant of Unix.

So how can the poor programmer cope with this multiplicity of line endings? In many languages she just has to do lots of tests and take different action per OS. In more modern languages, including Python, the language provides facilities for dealing with the mess for you. In the case of Python the assistance comes in the form of the `os` module which defines a variable called `linesep` which is set

to whatever the newline character is on the current operating system. This makes adding newlines easy, and `rstrip()` takes account of the OS when it does its work of removing them, so really the simple way to stay sane, so far as newlines are concerned is: always use `rstrip()` to remove newlines from lines read from a file and always add `os.linesep` to strings being written to a file.

That still leaves the awkward situation where a file is created on one OS and then processed on another, incompatible, OS and sadly, there isn't much we can do about that except to compare the end of the line with `os.linesep` to determine what the difference is.

Specifying Paths

This is more of an issue for Windows users than others although MacOS 9 users may bump into it occasionally too. As above each OS specifies paths to files using different characters to separate the drives, folders and files. The generic solution for this is again to use the `os` module which provides the `os.sep` variable to define the current platforms path separator character. In practice you won't need to use this very often since the path will likely be different for every machine anyway! So instead you will just enter the full path directly in a string, possibly once for each OS you are running on. But there is one big gotcha hiding in wait for Windows users...

You saw in the previous section that python treats the string `'\n'` as a newline character. That is it takes two characters and treats them as one. In fact there are a whole range of these special sequences beginning with back slash (\) including:

- `\n` - A new line
- `\r` - A carriage return
- `\t` - A horizontal tab
- `\v` - A vertical tab (sometimes means a new page)
- `\b` - A backspace
- `\0nn` - Any arbitrary octal character code. e.g. the code `\033` is the escape character (`ESC`)

This means that if we have a data file called `test.dat` and want to open it in Python by specifying a full Windows path we might expect this to work:

```
>>> f = open('C:\test.dat')
```

But Python will see the `\t` pair as a tab character and complain it cannot find a file called:
`C:    est.dat`.
So how do we get round this inconvenience? There are three solutions:

1. put an 'r' in front of the string. This tells Python to ignore any back slashes and treat it as a "raw" sting.

   ```
   >>> print "Hello\nWorld"
   >>> print r"Hello\nWorld"
   ```
2. Use forward slashes (/) instead of backslashes, Python and Windows will between them sort out the path for you. This has the added advantage of making your code portable to other operating systems too.
3. Use a double backslash(\\) since a double backslash character is seen by Python as a single backslash!

Thus any of the following will open our data file correctly:

```
>>> f = open(r'C:\test.dat')
>>> f = open('C:/test.dat')
>>> f = open('C:\\test.dat')
```

Appending data

One final twist in file processing is that you might want to append data to the end of an existing file. One way to do that would be to open the file for input, read the data into a list, append the data to the list and then write the whole list out to a new version of the old file. If the file is short that's not a problem but if the file is very large, maybe over 100Mb, then you could run out of memory to hold the list. Fortunately there's another mode `"a"` that we can pass to `open()` which allows us to append directly to an existing file just by writing. Even better, if the file doesn't exist it will open a new file just as if you'd specified `"w"`.

As an example, let's assume we have a log file that we use for capturing error messages. We don't want to delete the existing messages so we choose to append the error, like this:

```
def logError(msg):
   err = open("Errors.log","a")
   err.write(msg)
   err.close()
```

In the real world we would probably want to limit the size of the file in some way. A common technique is to create a filename based on the date, thus when the date changes we automatically create a new file and it is easy for the maintainers of the system to find the errors for a particular day and to archive away old error files if they are not needed. (Remember, from the menu example above, that the `time` module can be used to find out the current date.)

The Address Book Revisited

You remember the address book program we introduced during the Raw Materials topic and then expanded in the Talking to the User topic? Let's start to make it really useful by saving it to a file and, of course, reading the file at startup. We'll do this by writing some functions. So in this example we pull together several of the strands that we've covered in the last few topics.

The basic design will require a function to read the file at startup, another to write the file at the end of the program. We will also create a function to present the user with a menu of options and a separate function for each menu selection. The menu will allow the user to:

- Add an entry to the address book
- Remove an entry from the book
- Find and display an existing entry
- Quit the program

Loading the Address Book

```
import os
filename = "addbook.dat"

def readBook(book):
    if os.path.exists(filename):
```

```
        store = open(filename,'r')
        for line in store:
           name = line.rstrip()
           entry = store.next().rstrip()
           book[name] = entry
        store.close()
```

Note 1: We import the `os` module which we use to check that the file path actually exists before opening the file.

Note 2: We defined the filename as a module level variable so we can use it both in loading and saving the data.

Note 3: We use `rstrip()` to remove the new-line character from the end of the line. Also notice the `next()` operation to fetch the next line from the file within the loop. This effectively means we are reading two lines at a time as we progress through the loop.

Saving the Address Book

```
def saveBook(book):
    store = open(filename, 'w')
    for name,entry in book.items():
        store.write(name + '\n')
        store.write(entry + '\n')
    store.close()
```

Notice we need to add a newline character (`'\n'`) when we write the data. Also note that we write two lines for each entry, this mirrors the fact that we processed two lines when reading the file.

Getting User Input

```
def getChoice(menu, length):
    print( menu )
    prompt = "Select a choice(1-%d): "  % length
    choice = int( input(prompt) )
    return choice
```

Note: We receive a length parameter which tells us how many menu entries there are. This allows us to create a prompt that specifies the correct number range.

Adding an Entry

```
def addEntry(book):
    name =  input("Enter a name: ")
    entry = input("Enter street, town and phone number: ")
    book[name] = entry
```

Removing an entry

```
def removeEntry(book):
    name = input("Enter a name: ")
    del(book[name])
```

Finding an entry

```
def findEntry(book):
    name = input("Enter a name: ")
    if name in book:
       print( name, book[name] )
    else: print( "Sorry, no entry for: ", name )
```

Quitting the program

Actually I won't write a separate function for this, instead I'll make the quit option the test in my menu `while` loop. So the main program will look like this:

```
def main():
    theMenu = '''
    1) Add Entry
    2) Remove Entry
    3) Find Entry
    4) Quit and save
    '''
    theBook = {}
    readBook(theBook)
    choice = getChoice(theMenu, 4)
    while choice != 4:
        if choice == 1:
            addEntry(theBook)
        elif choice == 2:
            removeEntry(theBook)
        elif choice == 3:
            findEntry(theBook)
        else: print( "Invalid choice, try again" )
        choice = getChoice(theMenu, 4)
    saveBook(theBook)
```

Now the only thing left to do is call the `main()` function when the program is run, and to do that we use a bit of Python magic like this:

```
if __name__ == "__main__":
    main()
```

This mysterious bit of code allows us to use any python file as a module by `import`ing it, or as a program by running it. The difference is that when the program is imported, the internal variable `__name__` is set to the module name but when the file is run, the value of `__name__` is set to `"__main__"`. Sneaky, eh?

Now if you type all that code into a new text file and save it as addressbook.py, you should be able to run it from an OS prompt by typing:

```
C:\PROJECTS> python addressbook.py
```

Or just double click the file in Windows Explorer and it should start up in its own DOS window, and the window will close when you select the quit option.

Or, in Linux:

```
$ python addressbook.py
```

This 60 odd line program is typical of the sort of thing you can start writing for yourself. There are a couple of things we can do to improve it which I'll cover in the next section, but even as it stands it's a reasonably useful little tool.

VBScript and JavaScript

Neither VBScript nor JavaScript have native file handling capabilities. This is a security feature to ensure nobody can read your files when you innocently load a web page, but it does restrict their general usefulness. However, as we saw with reusable modules there is a way to do it using Windows Script Host. WSH provides a `FileSystem` object which allows any WSH language to read files. We will look at a JavaScript example in detail then show similar code in VBScript for comparison, but as before the key elements will really be calls to the WScript objects.

Before we can look at the code in detail it's worth taking time to describe the `FileSystem` *Object Model*. An Object Model is a set of related objects which can be used by the programmer. The WSH `FileSystem` object model consists of the `FSO` object, a number of `File` objects, including the `TextFile` object which we will use. There are also some helper objects, most notable of which is, for our purposes, the `TextStream` object. Basically we will create an instance of the `FSO` object, then use it to create our `TextFile` objects and from these in turn create `TextStream` objects to which we can read or write text. The `TextStream` objects themselves are what we actually read/write from the files.

Type the following code into a file called `testFiles.js` and run it using `cscript` as described in the earlier introduction to WSH.

Opening a file

To open a file in WSH we create an FSO object then create a TextFile object from that:

```
var fileName, fso, txtFile, outFile, line;

// Get file name
fso = new ActiveXObject("Scripting.FileSystemObject");
WScript.Echo("What file name? ");
fileName = WScript.StdIn.Readline();

// open inFile to read, outFile to write
inFile = fso.OpenTextFile(fileName, 1); // mode 1 = Read
fileName = fileName + ".BAK"
outFile = fso.CreateTextFile(fileName);
```

Reading and Writing a file

```
// loop over file till it reaches the end
while ( !inFile.AtEndOfStream ){
    line = inFile.ReadLine();
    WScript.Echo(line);
    outFile.WriteLine( line );
    }
```

Closing files

```
inFile.close();
outFile.close();
```

And in VBScript

Save the following as `testFIles.ws` and then run it using:

```
cscript testfiles.ws
```

Or alternatively, put the bit between the `script` tags into a file called `testFile.vbs` and run that instead. The `.ws` format allows you to mix JavaScript and VBScript code in the same file by simply using multiple `script` tags, should you want to...

```
<?xml version="1.0"?>

<job>
  <script type="text/vbscript">
      Dim fso, inFile, outFile, inFileName, outFileName
      Set fso = CreateObject("Scripting.FileSystemObject")

      WScript.Echo "Type a filename to backup"
      inFileName = WScript.StdIn.ReadLine
      outFileName = inFileName &amp; ".BAK"

      ' open the files
      Set inFile = fso.OpenTextFile(inFileName, 1)
      Set outFile = fso.CreateTextFile(outFileName)

      ' read the file and write to the backup copy
      While not inFile.AtEndOfStream
         line = inFile.ReadLine
         outFile.WriteLine(line)
      Wend

      ' close both files
      inFile.Close
      outFile.Close

      WScript.Echo inFileName &amp; " backed up to " &amp; outFileName
  </script>
</job>
```

Handling Non-Text Files

Handling text is one of the most common things that programmers do, but sometimes we need to process raw binary data too. This is very rarely done in VBScript or JavaScript so I will only be covering how Python does it.

Opening and Closing Binary Files

The key difference between text files and binary files is that text files are composed of *octets*, or bytes, of binary data whereby each byte represents a character and the end of the file is marked by a special byte pattern, known generically as *end of file*, or *eof*. A binary file contains arbitrary binary data and thus no specific value can be used to identify end of file, thus a different mode of operation is required to read these files. The end result of this is that when we open a binary file in Python (or indeed any other language) we must specify that it is being opened in binary mode or risk the data being read being truncated at the first `eof` character that Python finds in the data. The way we do this in Python is to add a 'b' to the mode parameter, like this:

```
binfile = file("aBinaryFile.bin","rb")
```

The only difference from opening a text file is the mode value of `"rb"`. You can use any of the other modes too, simply add a 'b': `"wb"` to write, `"ab"` to append.

Closing a binary file is no different to a text file, simply call the `close()` method of the open file object:

```
binfile.close()
```

Because the file was opened in binary mode there is no need to give Python any extra information, it knows how to close the file correctly.

Data Representation and Storage

Before we discuss how to access the data within a binary file we need to consider how data is represented and stored on a computer. All data is stored as a sequence of *bi*nary digi*t*s, or bits. These bits are grouped into sets of 8 or 16 called *bytes* or *words* respectively. (A group of 4 is sometimes called a nibble!) A byte can be any one of 256 different bit patterns and these are given the values 0-255.

The information we manipulate in our programs, strings, numbers etc must all be converted into sequences of bytes. Thus the characters that we use in strings are each allocated a particular byte pattern. There were originally several such *encodings*, but the most common is the *ASCII* (American Standard Coding for Information Interchange). Unfortunately pure ASCII only caters for 128 values which is not enough for non English languages. A new encoding standard known as *Unicode* has been produced, which can use data words instead of bytes to represent characters, and allows for over 65000 characters. (A more recent increase in spec has raised that to over a million!). These characters can then be *encoded* into a more compact data stream. One of the most common encodings is called *UTF-8* and it corresponds closely to the earlier ASCII coding such that every valid ASCII file is a valid UTF-8 file, although not necessarily the other way around. Unicode provides a number of different encodings each of which defines which bytes represent each Unicode numerical value (or code point in Unicode terms). If you are thinking that this is complicated you are right! It is the cost of building a global computer network that must work in lots of different languages. The good news if you are an English speaker is that for the most part you can ignore it! The exception is when reading data from a binary file, when you do need to know which encoding has been used to interpret the binary data successfully.

Python fully supports Unicode text and we can specify which particular *encoding* we want to apply by inserting a special comment at the top of a source file. A string of encoded characters is considered to be a *byte string* and has the type `bytes` whereas a string of unencoded text has the type `str`. The default encoding is usually UTF-8 (but, in theory at least, could be different!). I will not be covering the use of non UTF-8 encodings in this tutorial but there is an extensive "How-To" document on the Python web site.

The key thing to realize in all of this is that a binary stream of encoded unicode text is treated as a string of bytes and Python provides functions to convert between `bytes` and `str` values.

In the same way numbers need to be converted to binary codings too. For small integers it is simple enough to use the byte values directly, but for numbers larger than 255 (or negative numbers, or fractions) some additional work needs to be done. Over time various standard codings have emerged for numerical data and most programming languages and operating systems use these. For example, the American Institute of Electrical and Electronic Engineering (*IEEE*) have defined a number of codings for floating point numbers.

The point of all of this is that when we read a binary file we have to interpret the raw bit patterns into the correct *type* of data for our program. It is perfectly possible to interpret a stream of bytes that were originally written as a character string as a set of floating point numbers. Or course the original meaning will have been lost but the bit patterns could represent either. So when we read binary data it is extremely important that we convert it into the correct data type.

The *Struct* Module

To encode/decode binary data Python provides a module called `struct`, short for structure. `struct` works very much like the format strings we have been using to print mixed data. We provide a string representing the data we are reading and apply it to the byte stream that we are trying to interpret. We can also use struct to convert a set of data to a byte stream for writing, either to a binary file (or even a communications line!).

There are many different conversion format codes but we will only use the integer and string codes here. (You can look up the others on the Python documentation for the `struct` module.) The codes for integer and string are `i`, and `s` respectively. The `struct` format strings consist of sequences of codes with numbers pre-pended to indicate how many of the items we need. The exception is the `s` code where the prepended number means the length of the string. For example `4s` means a string of four characters (note 4 characters not 4 strings!).

Let's assume we wanted to write the address details, from our Address Book program above, as binary data with the street number as an integer and the rest as a string (This is a bad idea in practice since street "numbers" sometimes include letters!). The format string would look like:

```
'i34s' # assuming 34 characters in the address!
```

To cope with multiple address lengths we could write a function to create the binary string like this:

```
def formatAddress(address):
    # split breaks a string into a list of 'words'
    fields = address.split()
    number = int(fields[0])
    rest = ' '.join(fields[1:])
    format = "i%ds" % len(rest) #create the format string
    return struct.pack(format, number, rest)
```

So we used a string method - `split()` - (more on them in the next topic!) to split the address string into its parts, extract the first one as the number and then use another string method, `join` to join the remaining fields back together separated by spaces. The length of that string is the number we need in the `struct` format string so we use the `len()` function in conjunction with a normal format string to build a `struct` format string. Phew!

`formatAddress()` will return a sequence of bytes containing the binary representation of our address. Now that we have our binary data let's see how we can write that to a binary file and then read it back again.

Reading & Writing Using *Struct*

Let's create a binary file containing a single address line using the `formatAddress()` function defined above. We need to open the file for writing in `'wb'` mode, encode the data, write it to the file and then close the file. Let's try it:

```
import struct
f = file('address.bin','wb')
data = "10 Some St, Anytown, 0171 234 8765"
bindata = formatAddress(data)
print( "Binary data before saving: ", repr(bindata) )
f.write(bindata)
f.close()
```

You can check that the data is indeed in binary format by opening `address.bin` in notepad. The characters will be readable but the number will not look like 10! In fact it has disappeared! If you have an editor which can read binary files (e.g vim or emacs) and use that to open `address.bin` you will see that the start of the file has 4 bytes. The first of these may look like a newline character and the rest are zeros. Now it turns out that, just coincidentally, the numerical value of newline is 10! As we can show using Python:

```
>>> ord('\n')
10
```

The `ord()` function simply returns the numeric value of a given character. So the first 4 bytes are `10,0,0,0` in decimal (or `0xA,0x0,0x0,0x0` in hexadecimal, the system usually used to display binary data - since it is much more concise than using pure binary).

On a 32 bit computer an integer takes up 4 bytes. So the integer value '10' has been converted by the `struct` module into the 4 byte sequence `10, 0, 0, 0`. Now on intel micro-processors the byte sequence is to put the *least significant byte* first so that, reading it in reverse, gives us the true "binary" value: `0,0,0,10`.

Which is the integer value 10 expressed as 4 decimal bytes. The rest of the data is basically the original text string and so appears in its normal character format.

Be sure not to save the file from within Notepad since although Notepad can load some binary files it cannot save them as binary, it will try to convert the binary to text and can corrupt the data in the process! It is worth pointing out here that the file extension `.bin` that we used is purely for our convenience, it has no bearing on whether the file is binary or text format. Some Operating Systems use the extension to determine what programme they will use to open the file, but you can change the extension by simply renaming the file, the content will not change it will still be binary or text whichever it was originally. (You can prove this by renaming a text file in Windows to

`.exe` whereupon Windows will treat the file as an executable, but when you try to run it you will get an error because the text is not really executable binary code! If you now rename it back to `.txt` the file will open in Notepad exactly as it did before, the content has not been altered at all - in fact you could even have opened the text in Notepad while it was named as a `.exe` and it would have worked just as well!)

To read our binary data back again we need to open the file in `'rb'` mode, read the data into a sequence of bytes, close the file and finally unpack the data using a `struct` format string. The question is: how do we tell what the format string looks like? In general we would need to find the binary format from the file definition (there are several web sites which provide this information - for example Adobe publish the definition of their common PDF binary format). In our case we know it must be like the one we created in `formatAddress()`, namely `'iNs'` where N is a variable number. How do we determine the value of N?

The struct module provides some helper functions that return the size of each data type, so by firing up the Python prompt and experimenting we can find out how many bytes of data we will get back for each data type:

```
>>> import struct
>>> print struct.calcsize('i')
4
>>> print struct.calcsize('s')
1
```

Ok, we know that our data will comprise 4 bytes for the number and one byte for each character. So N will be the total length of the data minus 4. Let's try using that to read our file:

```
import struct
f = file('address.bin','rb')
data = f.read()
f.close()

fmtString = "i%ds" % (len(data) - 4)
number, rest = struct.unpack(fmtString, data)
rest = str(rest)
address = ' '.join((str(number),rest))

print( "Address after restoring data:", address )
```

Note: We had to convert `rest` to a string using the `str()` function since Python considered it to be of type `bytes` (see the sidebar above) which won't work with `join()`.

And that's it on binary data files, or at least as much as I'm going to say on the subject. As you can see using binary data introduces several complications and unless you have a very good reason I don't recommend it. But at least if you do need to read a binary file, you can do it (provided you know what the data represented in the first place of course!)

Random Access to Files

The last aspect of file handling that I'll consider is called *random access*. Random access means moving directly to a particular part of the file without reading all the intervening data. Some programming languages provide a special indexed file type that can do this very quickly but in most languages its built on top of the normal sequential file access that we have been using up till now.

The concept used is that of a cursor that marks the current position within the file, literally how many bytes we are from the beginning. We can move this cursor relative to its current position or relative to the start of the file. We can also ask the file to *tell* us where the cursor is currently.

By using a fixed linelength (perhaps by padding our data strings with spaces or some other character where necessary) we can jump to the start of a particular line by multiplying the length of a line by the number of lines. This is what gives the impression of random access to the data in the file.

Where am I?

To determine where we are in a file we can use the `tell()` method of a file. For example if I open a file and read three lines, I can then ask the file how far into the file I am.

Let's look at an example, first I will create a file with 5 lines of text all the same length (the equal length business isn't strictly necessary but it does make life easier!). Then I'll read three lines back and ask where we are. I'll then go back to the beginning, read one line then jump to the third line and print it, jumping over the second line. Like this:

```
# create 5 lines of 20 chars (+ \n)
testfile = open('testfile.txt','w')
for i in range(5):
    testfile.write(str(i) * 20 + '\n')
testfile.close()

# read 3 lines and ask where we are
testfile = open('testfile.txt','r')
for line in range(3):
    print( testfile.readline().strip() )
position = testfile.tell()
print( "At position: ", position, "bytes" )

# go back to the beginning
testfile.seek(0)
print( testfile.readline().strip() ) # repeat first line
lineLength = testfile.tell()
testfile.seek(2*lineLength)  # go to end of line 2
print( testfile.readline().strip() ) # the 3rd line
testfile.close()
```

Note the use of the `seek()` function to move the cursor. The default operation is to move it to the byte number specified, as shown here, but extra arguments can be provided that change the indexing method used. Also note that the value printed by the first `tell()` depends on the length of a newline on your platform, on my Windows XP PC it printed 66 indicating that the newline sequence is 2 bytes long. But since this is a platform specific value and I want to make my code portable I've used `tell()` again, after reading one line, to work out how long each line really is. These kind of "cunning ploys" are often necessary when dealing with platform specific issues!

Things to remember

- Open files before using them
- Files can usually only be read or written but not both at the same time
- Python's `readlines()` function reads all the lines in a file, while `readline()` only reads one line at a time, which may help save memory.
- However you don't usually need to use either since Python's `open` function works with `for` loops.
- Close files after use.
- Binary files need the mode flag to end in 'b' and you need to interpret the data after reading it - usually with the struct module.
- `tell()` and `seek()` enable pseudo-random access to sequential files

Previous Next Contents

If you have any questions or feedback on this page send me mail at: alan.gauld@btinternet.com

Manipulating Text

What will we cover?
• How to split lines of text into character groups • How to search for strings of text within other strings • How to replace text within a string • How to change case of characters

Handling text is one of the most common things that programmers do. As a result there are lots of specific tools in most programming languages to make this easier. In this section we will look at some of these and how we might use them in performing typical programming tasks.

Some of the most common tasks that we can do when working with text are:

- splitting lines of text into character groups
- searching for strings of text within other strings
- replacing text within a string
- changing case of characters

We will look at how to do each of these tasks using Python and then briefly consider how VBScript and JavaScript handle text processing.

In Python we use string *methods* to manipulate text strings. You might recall, from the Raw Materials topic, that methods are like functions attached to data. We can access the methods using the same dot notation we use to access functions in a module, but instead of using a module name we use the data itself. Let's see how that works.

Splitting strings

The first task we consider is how to split a string into its constituent parts. This is often necessary when processing files since we tend to read a file line by line, but the data may well be contained within segments of the line. An example of this is our Address Book example, where we might want to access the individual fields of the entries rather than just print the whole entry.

The Python method we use for this is called `split()` and it is used like this:

```
>>> aString = "Here is a (short) String"
>>> print( aString.split() )
['Here', 'is', 'a', '(short)', 'String']
```

Notice we get a list back containing the words within `aString` with all the spaces removed. The default separator for `''.split()` is *whitespace* (ie. tabs, newlines and spaces). Let's try using it again but with an opening parenthesis as the separator:

```
>>> print( aString.split('(') )
['Here is a ', 'short) String']
```

Notice the difference? There are only two elements in the list this time and the opening parenthesis has been removed from the front of `'short)'`. That's an important point to note about `''.split()`, that it removes the separator characters. Usually that's what we want, but just occasionally we'll wish it hadn't!

There is also a `''.join()` method which can take a list (or indeed any other kind of sequence) of strings and join them together. One confusing feature of `''.join()` is that it uses the string on which we call the method as the joining characters. You'll see what I mean from this example:

```
>>> lst = ['here','is','a','list','of','words']
>>> print( '-+-'.join(lst) )
here-+-is-+-a-+-list-+-of-+-words
>>> print( ' '.join(lst) )
here is a list of words
```

It sort of makes sense when you think about it, but it does look wierd when you first see it.

Counting words

Let's revisit that word counting program I mentioned in the functions topic. Recall the *Pseudo Code* looked like:

```
def numwords(aString):
    list = split(aString) # list with each element a word
    return len(list) # return number of elements in list

for line in file:
    total = total + numwords(line) # accumulate totals for each line

print( "File had %d words" % total )
```

Now we know how to get the lines from the file let's consider the body of the `numwords()` function. First we want to create a list of words in a line. That's nothing more than applying the default `''.split()` method. Referring to the Python documentation we find that the builtin function `len()` returns the number of elements in a list, which in our case should be the number of words in the string - exactly what we want.

So the final code looks like:

```
def numwords(aString):
    lst = aString.split() # split() is a method of the string object aString
    return len(lst) # return number of elements in the list

inp = file("menu.txt","r")
total = 0  # initialize to zero; also creates variable

for line in inp:
    total = total + numwords(line)  # accumulate totals for each line

print( "File had %d words" % total )

inp.close()
```

That's not quite right of course because it counts things like an ampersand character as a word (although maybe you think it should...). Also, it can only be used on a single file (menu.txt). But it's not too hard to convert it to read the filename from the command line (`argv[1]`) or via `input()` as we saw in the Talking to the user section. I'll leave that as an exercise for the reader.

Searching Text

The next common operation we will look at is searching for a sub-string within a longer string. This is again supported by a Python string method, this time called `''.find()` It's basic use is quite simple, you provide a search string and if Python finds it within the main string it returns the index of the first character of the substring, if it doesn't find it, it returns -1:

```
>>> aString = "here is a long string with a substring inside it"
>>> print( aString.find('long') )
10
>>> print( aString.find('oxen') )
-1
>>> print( aString.find('string') )
15
```

The first two examples are straightforward, the first returns the index of the start of `'long'` and the second returns `-1` because `'oxen'` does not occur inside `aString`. The third example throws up an interesting point, namely that find only locates the *first* occurrence of the search string, but what do we do if the search string occurs more than once in the original string?

One option is to use the index of the first occurrence to chop the original string into two pieces and search again. We keep doing this until we get a -1 result. Like this:

```
aString = "Bow wow says the dog, how many ow's are in this string?"
temp = aString[:] # use slice to make a copy
count = 0
index = temp.find('ow')
while index != -1:
    count += 1
    temp = temp[index + 1:]  # use slicing
    index = temp.find('ow')

print( "We found %d occurrences of 'ow' in %s" % (count, aString) )
```

Here we just counted occurrences, but we could just as well have collected the index results into a list for later processing.

The `find()` method can speed this process up a little by using a one of its extra optional parameters. That is, a start location within the original string:

```
aString = "Bow wow says the dog, how many o's are in this string?"
count = 0
index = aString.find('ow')  # use default start
while index != -1:
    count += 1
    start = index + 1
    index = aString.find('ow', start)  # set new start

print( "We found %d occurrences of 'ow' in %s" % (count, aString) )
```

This solution removes the need to create a new string each time, which can be a slow process if the string is long. Also, if we know that the substring will definitely only be within the first so many characters (or we aren't interested in later occurrences) we can specify both a start and stop value, like this:

```
>>>  # limit search to the first 20 chars
>>> aString = "Bow wow says the dog, how many ow's are in the string?"
```

```
>>> print( aString.find('the',0,20) )
```

To complete our discussion of searching there are a couple of nice extra methods that Python provides to cater for common search situations, namely `''.startswith()` and `''.endswith()`. From the names alone you probably can guess what these do. They return True or False depending on whether the original string starts with or ends with the given search string, like this:

```
>>> print( "Python rocks!".startswith("Perl") )
False
>>> print( "Python rocks!".startswith('Python') )
True
>>> print( "Python rocks!".endswith('sucks!') )
False
>>> print( "Python rocks!".endswith('cks!') )
True
```

Notice the boolean result. After all, you already know where to look if the answer is True! Also notice that the search string doesn't need to be a complete word, a substring is fine. You can also provide a `start` and `stop` position within the string, just like `''.find()` to effectively test for a string at any given location within a string. This is not a feature that is used much in practice.

And finally for a simple test of whether a substring exists anywhere within another string you can use the Python `in` operator, like this:

```
>>> if 'foo' in 'foobar': print( 'True' )
True
>>> if 'baz' in 'foobar': print( 'True' )
>>> if 'bar' in 'foobar': print( 'True' )
True
```

That's all I'll say about searching for now, let's look at how to replace text next.

Replacing text

Having found our text we often want to change it to something else. Again the Python string methods provide a solution with the `''.replace()` method. It takes two arguments: a search string and a replacement string. The return value is the new string as a result of the replacement.

```
>>> aString = "Mary had a little lamb, its fleece was dirty!"
>>> print( aString.replace('dirty','white') )
"Mary had a little lamb, its fleece was white!"
```

One interesting difference between `''.find()` and `''.replace` is that replace, by default, replaces **all** occurrences of the search string, not just the first. An optional `count` argument can limit the number of replacements:

```
>>> aString = "Bow wow wow said the little dog"
>>> print( aString.replace('ow','ark') )
Bark wark wark said the little dog
>>> print( aString.replace('ow','ark',1) ) # only one
Bark wow wow said the little dog
```

It is possible to do much more sophisticated search and replace operations using something called a *regular expression*, but they are much more complex and get a whole topic to themselves in the "Advanced" section of the tutorial.

Changing the case of characters

One final thing to consider is converting case from lower to upper and vice-versa. This isn't such a common operation but Python does provide some helper methods to do it for us:

```
>>> print( "MIXed Case".lower() )
mixed case
>>> print( "MIXed Case".upper() )
MIXED CASE
>>> print( "MIXed Case".swapcase() )
mixED cASE
>>> print( "MIXed Case".capitalize() )
Mixed case
>>> print( 'MIXed Case'.title() )
Mixed Case
>>> print( "TEST".isupper() )
True
>>> print( "TEST".islower() )
False
```

Note that `''.capitalize()` capitalizes the entire string not each word within it - that's `title()`'s job!. Also note the two test functions (or *predicates*) `''.isupper()` and `''.islower()`. Python provides a whole bunch of these predicate functions for testing strings, other useful tests include: `''.isdigit()`, `''.isalpha()` and `''.isspace()`. The last checks for all kinds of whitespace not just literal space characters!

We will be using many of these string methods as we progress through the tutorial, and in particular the Grammar Counter case study uses several of them.

Text handling in VBScript

Because VBScript descends from BASIC it has a wealth of builtin string handling functions. In fact in the reference documentation I counted at least 20 functions or methods, not counting those that are simply there to handle Unicode characters.

What this means is that we can pretty much do all the things we did in Python using VBScript too. I'll quickly run through the options below:

Splitting text

We start with the `Split` function:

```
<script type="text/vbscript">
Dim s
Dim lst
s = "Here is a string of words"
lst = Split(s) ' returns an array
MsgBox lst(1)
</script>
```

As with Python you can add a separator value if the default whitespace separation isn't what you need.

Also as with Python there is a `Join` function for reversing the process.

Searching for and replacing text

Searching is done with `InStr`, short for "In String", obviously.

```
<script type="text/vbscript">
Dim s,n
s = "Here is a long string of text"
n = InStr(s, "long")
MsgBox "long is found at position: " & CStr(n)
</script>
```

The return value is normally the position within the original string that the substring starts. If the substring is not found then zero is returned (this isn't a problem because VBScript starts its indices at 1, so zero is not a valid index). If either string is a `Null` a `Null` is returned, which makes testing error conditions slightly more tricky with a combined test required.

As with Python we can specify a sub range of the original string to search, using a start value, like this:

```
<script type="text/vbscript">
Dim s,n
s = "Here is a long string of text"
n = InStr(6, s, "long") ' start at position 6
If n = 0 or n = Null Then ' check for errors
   MsgBox "Invalid string found"
Else
   MsgBox "long is found at position: " & CStr(n)
End If
</script>
```

Unlike Python we can also specify whether the search should be case-sensitive or not, the default is case-sensitive.

Replacing text is done with the `Replace` function. Like this:

```
<script type="text/vbscript">
Dim s
s = "The quick yellow fox jumped over the log"
MsgBox Replace(s, "yellow", "brown")
</script>
```

We can provide an optional final argument specifying how many occurrences of the search string should be replaced, the default is all of them. We can also specify a start position as for `InStr` above.

Changing case

Changing case in VBScript is done with `UCase` and `LCase`, there is no equivalent of Python's `capitalize` or `title` methods.

```
<script type="text/vbscript">
Dim s
s = "MIXed Case"
MsgBox LCase(s)
```

```
MsgBox UCase(s)
</script>
```

And that's all I'm going to cover in this tutorial, if you want to find out more check the VBScript help file for the list of functions.

Text handling in JavaScript

JavaScript is the least well equipped for text handling of our three languages. Even so, the basic operations are catered for to some degree, it is only in the number of "bells & whistles" that JavaScript suffers in comparison to VBScript and Python. JavaScript compensates somewhat for its limitations with strong support for *regular expressions* (which we cover in a later topic) and these extend the apparently primitive functions quite significantly, but at the expense of some added complexity.

Like Python JavaScript takes an object oriented approach to string manipulation, with all the work being done by methods of the `String` class.

Splitting Text

Splitting text is done using the `split` method:

```
<script type="text/javascript">
var aList, aString = "Here is a short string";
aList = aString.split(" ");
document.write(aList[1]);
</script>
```

Notice that JavaScript requires the separator character to be provided, there is no default value. The separator is actually a regular expression and so quite sophisticated split operations are possible.

Searching Text

Searching for text in JavaScript is done via the `search()` method:

```
<script type="text/javascript">
var aString = "Round and Round the ragged rock ran a rascal";
document.write( "ragged is at position: " + aString.search("ragged"));
</script>
```

Once again the search string argument is actually a regular expression so the searches can be very sophisticated indeed. Notice, however, that there is no way to restrict the range of the original string that is searched by passing a start position (although this can also be simulated using regular expression tricks).

JavaScript provides another search operation with slightly different behaviour called `match()`, I don't cover the use of `match` here.

Replacing Text

To do a replace operation we use the `replace()` method.

```
<script type="text/javascript">
var aString = "Humpty Dumpty sat on a cat";
```

```
document.write(aString.replace("cat","wall"));
</script>
```

And once again the search string can be a regular expression, you can begin to see the pattern I suspect! The replace operation replaces all instances of the search string and, so far as I can tell, there is no way to restrict that to just one occurence without first splitting the string and then joining it back together.

Changing case

Changing case is performed by two functions: `toLowerCase()` and `toUpperCase()`

```
<script type="text/javascript">
var aString = "This string has Mixed Case";
document.write(aString.toLowerCase()+ "<BR>");
document.write(aString.toUpperCase()+ "<BR>");
</script>
```

There is very little to say about this pair, they do a simple job simply. JavaScript, unlike the other languages we consider provides a wealth of special text functions for processing HTML, this revealing it's roots as a web programming language. We don't consider these here but they are all described in the standard documentation.

That concludes our look at text handling, hopefully it has given you the tools you need to process any text you encounter in your own projects. One final word of advice: always check the documentation for your language when processing text, there are often powerful tools included for this most fundamental of programming tasks.

Things to remember
• Text processing is a common operation with powerful support built-in to most languages • The most common tasks are splitting text, searching for and replacing text and changing case • Each language provides different levels of support but the three basic operations are nearly always available.

Previous Next Contents

If you have any questions or feedback on this page send me mail at: alan.gauld@btinternet.com

Handling Errors

What will we cover?
• A short history of error handling • Two techniques for handling errors • Defining and raising errors in our code for others to catch

A Brief History of Error Handling

Error handling is the process of catching the errors generated by our program and hiding them from our users. It doesn't matter too much if we, as programmers get exposed to a Python error message - we are supposed to understand all that techno speak. But our users are probably not programmers and they want nice, easy to understand messages when things go wrong, and ideally, they want us to catch the error and fix it without them ever knowing about it!

And that's where error handling comes in. Almost every language provides a mechanism for catching errors as they occur, finding out what went wrong and, if possible, taking appropriate action to fix the problem. Over time there have been a number of different approaches adopted to do this and we tackle the subject by following the historical development of the technology. In that way you can hopefully appreciate why the new methods have been introduced. At the end of the topic you should be able to write user friendly programs that never allow a hint of a Python error message to be seen by your users.

VBScript is by far the most bizarre of our three languages in the way it handles errors. The reason for this is that it is built on a foundation of BASIC which was one of the earliest programming languages (around 1963) and VBScript error handling is one place where that heritage shines through. For our purposes that's not a bad thing because it gives me the opportunity to explain why VBScript works as it does by tracing the history of error handling from BASIC through Visual Basic to VBScript. After that we will look at a much more modern approach as exemplified in both JavaScript and Python.

In traditional BASIC, programs were written with line numbers to mark each one of code. Transferring control was done by jumping to a specific line using a statement called GOTO (we saw an example of this in the Branching topic). Essentially this was the only form of control possible. In this environment a common mode of error handling was to declare an `errorcode` variable that would store an integer value. Whenever an error occurred in the program the `errorcode` variable would be set to reflect the problem - couldn't open a file, type mismatch, operator overflow etc

This led to code that looked like this fragment out of a fictitious program:

```
1010 LET DATA = INPUT FILE
1020 CALL DATA_PROCESSING_FUNCTION
1030 IF NOT ERRORCODE = 0 GOTO 5000
1040 CALL ANOTHER_FUNCTION
1050 IF NOT ERRORCODE = 0 GOTO 5000
1060 REM CONTINUE PROCESSING LIKE THIS
...
5000 IF ERRORCODE = 1 GOTO 5100
5010 IF ERRORCODE = 2 GOTO 5200
5020 REM MORE IF STATEMENTS
...
5100 REM HANDLE ERROR CODE 1 HERE
...
5200 REM HANDLE ERROR CODE 2 HERE
```

As you can see almost half of the main program is concerned with detecting whether an error occurred. Over time a slightly more elegant mechanism was introduced whereby the detection of errors and their handling was partially taken over by the language interpreter, this looked like:

```
1010 LET DATA = INPUTFILE
1020 ON ERROR GOTO 5000
1030 CALL DATA_PROCESSING_FUNCTION
1040 CALL ANOTHER_FUNCTION
...
5000 IF ERRORCODE = 1 GOTO 5100
5010 IF ERRORCODE = 2 GOTO 5200
```

This allowed a single line to indicate where the error handling code would reside. It still required the functions which detected the error to set the `ERRORCODE` value but it made writing (and reading!) code much easier.

So how does this affect us? Quite simply Visual Basic still provides this form of error handling although the line numbers have been replaced with more human friendly labels. VBScript as a descendant of Visual Basic provides a severely cut down version of this. In effect VBScript allows us to choose between handling the errors locally or ignoring errors completely.

To ignore errors we use the following code:

```
On Error Goto 0  ' 0 implies go nowhere
SomeFunction()
SomeOtherFunction()
....
```

To handle errors locally we use:

```
On Error Resume Next
SomeFunction()
If Err.Number = 42 Then
   ' handle the error here
SomeOtherFunction()
...
```

This seems slightly back to front but in fact simply reflects the historical process as described above.

The default behavior is for the interpreter to generate a message to the user and stop execution of the program when an error is detected. This is what happens with `GoTo 0` error handling, so in effect `GoTo 0` is a way of turning off local control and allowing the interpreter to function as usual.

`Resume Next` error handling allows us to either pretend the error never happened, or to check the Error object (called `Err`) and in particular the number attribute (exactly like the early errorcode technique). The `Err` object also has a few other bits of information that might help us to deal with the situation in a less catastrophic manner than simply stopping the program. For example we can find out the source of the error, in terms of an object or function etc. We can also get a textual description that we could use to populate an informational message to the user, or write a note in a log file. Finally we can change error type by using the `Raise` method of the `Err` object. We can also use `Raise` to generate our own errors from within our own Functions.

As an example of using VBScript error handling lets look at the common case of trying to divide by zero:

```
<script type="text/vbscript">
Dim x,y,Result
x = Cint(InputBox("Enter the number to be divided"))
y = CINt(InputBox("Enter the number to divide by"))
On Error Resume Next
Result = x/y
If Err.Number = 11 Then ' Divide by zero
   Result = Null
End If
On Error GoTo 0 ' turn error handling off again
If VarType(Result) = vbNull Then
   MsgBox "ERROR: Could not perform operation"
Else
   MsgBox CStr(x) & " divided by " & CStr(y) & " is " & CStr(Result)
End If
</script>
```

Frankly that's not very nice and while an appreciation of ancient history may be good for the soul, modern programming languages, including both Python and JavaScript, have much more elegant ways to handle errors, so let's look at them now.

Error Handling in Python

Exception Handling

In recent programming environments an alternative way of dealing with errors known as *exception handling* works by having functions *throw* or *raise* an *exception*. The system then forces a jump out of the current block of code to the nearest exception handling block. The system provides a default handler which *catches* all exceptions which have not already been handled elsewhere and usually prints an error message then exits.

One big advantage of this style of error handling is that the main function of the program is much easier to see because it is not mixed up with the error handling code, you can simply read through the main block without having to look at the error code at all.

Let's see how this style of programming works in practice.

Try/Except

The exception handling block is coded rather like an `if...then...else` block:

```
try:
   # program logic goes here
except ExceptionType:
   # exception processing for named exception goes here
except AnotherType:
   # exception processing for a different exception goes here
else:
   # here we tidy up if NO exceptions are raised
```

Python attempts to execute the statements between the `try` and the first `except` statement. If it encounters an error it will stop execution of the `try` block and jump down to the `except` statements. It will progress down the `except` statements until it finds one which matches the error (or *exception*) type and if it finds a match it will execute the code in the block immediately following that exception. If no matching except statement is found, the error is propagated up to the next level of the program until, either a match is found or the top level Python interpreter catches the error, displays an error message and stops program execution - this is what we have seen happening in our programs so far.

If no errors are found in the `try` block then the final `else` block is executed although, in practice, this feature is rarely used. Note that an `except` statement with no specific error type will catch all error types not already handled. In general this is a bad idea, with the exception of the top level of your program where you may want to avoid presenting Python's fairly technical error messages to your users, you can use a general except statement to catch any uncaught errors and display a friendly "shutting down" type message.

It is worth noting that Python provides a `traceback` module which enables you to extract various bits of information about the source of an error, and this can be useful for creating log files and the like. I won't cover the traceback module here but, if you need it, the standard module documentation provides a full list of the available features.

Let's look at a real example now, just to see how this works:

```
value = input("Type a divisor: ")
try:
   value = int(value)
   print( "42 / %d = %d" % (value, 42/value) )
except ValueError:
   print( "I can't convert the value to an integer" )
except ZeroDivisionError:
   print( "Your value should not be zero" )
except:
   print( "Something unexpected happened" )
else: print( "Program completed successfully" )
```

If you run that and enter a non-number, a string say, at the prompt, you will get the `ValueError` message, if you enter `0` you will get the `ZeroDivisionError` message, if you hit `Ctrl-C` it will raise a `KeyboardInterrupt` exception and you'll see the "Something unexpected..." message and, finally, if you enter a valid number you will get the result plus the "Program completed" message.

Try/Finally

There is another type of 'exception' block which allows us to tidy up after an error, it's called a `try...finally` block and typically is used for closing files, flushing buffers to disk etc. The `finally` block is *always* executed last regardless of what happens in the `try` section.

```
try:
   # normal program logic
finally:
   # here we tidy up regardless of the
   # success/failure of the try block
```

This becomes very powerful when combined with a `try/except` block. It looks like this:

```
print( "Program starting" )
try:
   data = open("data.dat")
   print( "data file opened" )
   value = int(data.readline().split()[2])
   print( "The calculated value is %s" % (value/(42-value)) )
except ZeroDivisionError:
   print( "Value read was 42" )
finally:
```

```
   data.close()
   print( "data file closed" )

print( "Program completed" )
```

Note: The data file should contain a line with a number in the 3rd field, something like:

```
Foo bar 42
```

In this case the data file will always be closed regardless of whether an exception is raised in the try/except block or not. Note that this is different behavior to the `else` clause of `try/except` because it only gets called if no exception is raised, and equally simply putting the code outside the `try/except` block would mean the file was not closed if the exception was anything other than a `ZeroDivisionError`. Only adding a `finally` block ensures that the file is *always* closed.

Also notice that I put the `open()` statement inside the `try/except` block. If I'd actually wanted to catch a file open error all I'd need to do is add another `except` block for an `IOError`. Why not try that yourself then try opening a non-existent file to see it in action?

Generating Errors

What happens when we want to generate exceptions for other people to catch, in a module say? In that case we use the `raise` keyword in Python:

```
numerator = 42
denominator = int( input("What value will I divide 42 by?") )
if denominator == 0:
   raise ZeroDivisionError
```

This raises a `ZeroDivisionError` exception which can be caught by a `try/except` block. To the rest of the program it looks exactly as if Python had generated the error internally. Another use of the `raise` keyword is to propagate an error to a higher level in the program from within an except block. For example we may want to take some local action, log the error in a file say, but then allow the higher level program to decide what ultimate action to take. It looks like this:

```
def div127by(datum):
    try:
      return 127/(42-datum)
    except ZeroDivisionError:
      logfile = open("errorlog.txt","w")
      logfile.write("datum was 42\n")
      logfile.close()
      raise

try:
   div127by(42)
except ZeroDivisionError:
   print( "You can't divide by zero, try another value" )
```

Notice how the function `div127by()` catches the error, logs a message in the error file and then passes the exception back up for the outer `try/except` block to deal with by calling `raise` with no specified error object.

User Defined Exceptions

We can also define our own exception types for even finer grained control of our programs. We do this by defining a new exception class (we briefly looked at defining classes in the Raw Materials topic and will look at it in more detail in the Object Oriented Programming topic later in the tutorial). Usually an exception class is trivial and contains no content of its own, we simply define it as a sub-class of `Exception` and use it as a kind of "smart label" that can be detected by `except` statements. A short example will suffice here:

```
>>> class BrokenError(Exception): pass
...
>>> try:
...   raise BrokenError
... except BrokenError:
...   print( "We found a Broken Error" )
...
```

Note that we use a naming convention of adding "Error" to the end of the class name and that we *inherit* the behavior of the generic `Exception` class by including it in parentheses after the name - we'll learn all about inheritance in the OOP topic.

One final point to note on raising errors. Up until now we have quit our programs by importing sys and calling the `exit()` function. Another method that achieves exactly the same result is to raise the `SystemExit` error, like this:

```
>>> raise SystemExit
```

The main advantage being that we don't need to `import sys` first.

JavaScript

JavaScript handles errors in a very similar way to Python, using the keywords `try`, `catch` and `throw` in place of Python's `try`, `except` and `raise`.

We'll take a look at some examples but the principles are exactly the same as in Python. Recent versions of JavaScript have introduced the `finally` construct and JavaScript's `finally` clause can also be combined with `try/catch` in a single construct; see the JavaScript documentation for details.

Catching errors

Catching errors is done by using a `try` block with a set of `catch` statements, almost identically to Python:

```
<script type="text/javascript">
try{
   var x = NonExistentFunction();
   document.write(x);
}
catch(err){
   document.write("We got an error in the code");
}
</script>
```

One big difference is that you only get to use one `catch` statement per `try` construct, you have to examine the error passed to see what kind it is inside the `catch` block. This is, in my view, a bit more messy than Python's multiple `except` style based on exception type. You can see a basic example of testing the error value in the code below.

Raising errors

Similarly we can raise errors by using the `throw` keyword just as we used the `raise` keyword in Python. We can also create our own error types in JavaScript as we did in Python but a much easier method is just to use a string.

```
<script type="text/javascript">
try{
   throw("New Error");
}
catch(e){
   if (e == "New Error")
      document.write("We caught a new error");
   else
      document.write("An unexpected error found");
}
</script>
```

And that's all I'll say about error handling. As we go through the more advanced topics coming up you will see error handling in use, just as you will see the other basic concepts such as sequences, loops and branches. In essence you now have all of the tools at your disposal that you need to create powerful programs. It might be a good idea to take some time out to try creating some programs of your own, just a couple, to try to sound these ideas into your head before we move on to the next set of topics. Here are a few sample ideas:

- A simple game such as OXO or Hangman
- A basic database, maybe based on our address book, for storing details of your video, DVD or CD collection.
- A diary utility that will let you store important events or dates and, if you feel really keen, that automatically pops up a reminder.

To complete any of the above you will need to use all of the language features we have discussed and probably a few of the language modules too. Remember to keep checking the documentation, there will probably be quite a few tools that will make the job easier if you look for them. Also don't forget the power of the Python >>> prompt. Try things out there until you understand how they work then transfer that knowledge into your program - it's how the professionals do it! Most of all, have fun!

See you in the Advanced section :-)

Things to remember
• Check VBScript error codes using an `if` statement • Catch exceptions with a Python `except` or JavaScript `catch` clause • Generate exceptions using the Python `raise` or JavaScript throw keyword • Error types can be a class in Python or a simple string in JavaScript

Previous Next Contents

If you have any questions or feedback on this page send me mail at: alan.gauld@btinternet.com

Namespaces

What will we cover?

- The meaning of namespace and scope and why they are important
- How namespaces work in Python
- Namespaces in VBScript and JavaScript

Introduction

What's a *namespace*? I hear you ask. Well, it's kinda hard to explain. Not because they are especially complicated, but because every language does them differently. The concept is pretty straightforward, a namespace is a space or region, within a program, where a name (of a variable, function, class etc) is valid. We actually use this idea in everyday life. Suppose you work in a big company and there is a colleague called Joe. In the accounts department there is another guy called Joe who you see occasionally but not often. In that case you refer to your colleague as "Joe" and the other one as "Joe in Accounts". You also have a colleague called Susan and there is another Susan in Engineering with whom you work closely. When referring to them you might say "Our Susan" or "Susan from Engineering". Do you see how you use the department name as a qualifier? That's what namespaces do in a program, they tell both programmers and the translator which of several identical names is being referred to.

They came about because early programming languages (like BASIC) only had *Global Variables*, that is, ones which could be seen throughout the program - even inside functions. This made maintenance of large programs difficult since it was easy for one bit of a program to modify a variable without other parts of the program realizing it - this was called a *side-effect*. To get round this, later languages (including modern BASICs) introduced the concept of namespaces. (C++ has taken this to extremes by allowing the programmer to create their own namespaces anywhere within a program. This is useful for library creators who might want to keep their function names unique when mixed with libraries provided by another supplier.)

Another term used to describe a namespace is *scope*. The scope of a name is the extent of a program whereby that name can be unambiguously used, for example inside a function or a module. A name's namespace is exactly the same as it's scope. There are a few very subtle differences between the terms but only a Computer Scientist pedant would argue with you, and for our purposes namespace and scope are identical.

Python's approach

In Python every module creates it's own namespace. To access those names we have to either precede them with the name of the module or explicitly import the names we want to use into our module's namespace. Nothing new there, we've been doing it with the `sys` and `time` modules already. (A class definition also creates its own namespace. Thus, to access a method or property of a class, we need to use the name of the instance variable or the class name first. We'll talk a lot more about that in the OOP topic.)

In Python there are a total of five possible namespaces (or *scopes*):

1. Built in scope - names defined within Python itself, these are always available from anywhere in your program.

2. Module scope - names defined, and therefore visible within a file or module, confusingly this is referred to as *global* scope in Python whereas global normally means visible from *anywhere* in other languages.
3. Local scope - names defined within a function or a class method
4. Class scope - names defined inside classes, we'll touch on these in the OOP topic.
5. Nested scope - a slightly complex topic which you can pretty much ignore!

Let's take a look at a piece of code that includes examples of all of these (except class and nested):

```
def square(x):
   return x*x

data = int(input('Type a number to be squared: '))
print( data, 'squared is: ', square(data) )
```

The following table lists each name and the scope to which it belongs:

Name	Namespace
square	Module/global
x	local (to square)
data	Module/global
int	built-in
input	built-in
print	built-in

Note that we don't count `def` or `return` as names because they are *keywords* or, part of the language definition, if you try to use a keyword as the name of a variable or function you will get an error.

So far so good. Now how does this come together when variables in different namespaces have the same name? Or when we need to reference a name that is not in the current namespace?

Accessing Names outside the Current Namespace

Here we look in more detail at exactly how Python locates names, even when the names we are using are not in the immediate namespace. It is resolved as follows, Python will look:

1. within it's local namespace (the current function),
2. within the module scope (the current file),
3. the built-in scope.

But what if the name is in a different module? Well, we `import` the module, as we've already seen many times in the tutorial. Importing the module actually makes the module *name* visible in our module namespace. We can then use the module name to access the variable names within the module using our familiar `module.name` style. This explains why, in general, it is not a good idea to import all the names from a module into the current file: there is a danger that a module name will be the same as one of your variables and one of them will mask the other causing strange behavior in the program.

For example let's define two modules, where the second imports the first:

```
##### module first.py #########
spam = 42
```

```
def print42(): print( spam )
################################


##### module second.py ########
from first import *  # import all names from first

spam = 101   # create spam variable, hiding first's version
print42()    # what gets printed? 42 or 101?

################################
```

If you thought it would print 101 then you were wrong (and I admit I expected that when I first wrote the example!). The reason why it prints 42 instead has to do with the definition of a variable in Python as we described it away back in the Raw Materials topic. Recall that a name is simply a label used to reference an object. Now in the first module the name `print42` refers to the function object defined in the module (if this sounds odd there's more explanation in the advanced topic Functional Programming where it discusses something called a *lambda expression*). So although we imported the name into our module we did not import the function, which still refers to its own module's version of `spam`. Thus when we created our new spam variable it has no effect on the function referred to by the name `print42`

All of that confusion should serve to illustrate why, although it's more typing, it is much safer to access names in imported modules using the dot notation. There are a few modules, such as Tkinter which we'll meet later, which are commonly used by importing all of the names, but they are written in such a way to minimize the risk of name conflicts, although the risk always exists and can create very hard to find bugs.

Finally there is another safe way to import a single name from a module, like this:

```
from sys import exit
```

Here we only bring the `exit` function into the local namespace. We cannot use any other `sys` names, not even `sys` itself!

Avoiding Name Clashes

If a function refers to a variable called X and there exists an X within the function (local scope) then that is the one that will be seen and used by Python. It's the programmer's job to avoid name clashes such that a local variable and module variable of the same name are not both required in the same function - the local variable will mask the module name.

There is no problem if we just want to read a global variable inside a function, Python simply looks for the name locally, and not finding it will look globally (and if need be at the built-in namespace too). The problem arises when we want to assign a value to a global variable. That would normally create a new local variable inside the function. So, how can we assign a value to a global variable without creating a local variable of the same name? We can achieve this by use of the `global` keyword:

```
var = 42
def modGlobal():
   global var  # prevent creation of a local var
   var = var - 21
```

```
def modLocal():
   var = 101

print( var )   # prints 42
modGlobal()
print( var )  # prints 21
modLocal()
print( var )  # still prints 21
```

Here we see the global variable being changed by the `modGlobal` function but not changed by the `modLocal` function. The latter simply created its own internal variable and assigned it a value. At the end of the function that variable was garbage collected and its existence was unseen at the module level.

In general you should minimize the use of 'global' statements, it's usually better to pass the variable in as a parameter and then return the modified variable. Here is the `modGlobal` function above rewritten to avoid using a `global` statement:

```
var = 42
def modGlobal(aVariable):
    return aVariable - 21

print( var )
var = modGlobal(var)
print( var )
```

In this case we assign the return value from the function to the original variable while also passing it in as an argument. The result is the same but the function now has no dependencies on any code outside itself - this makes it much easier to reuse in other programs. It also makes it much easier to see how the global value gets changed - we can see the explicit assignment taking place.

We can see all of this at work in this example (which does nothing very useful, it is purely about illustrating the points made so far!):

```
# variables with module scope
W = 5
Y = 3

#parameters are like function variables
#so X has local scope
def spam(X):

   #tell function to look at module level and not create its own W
   global W

   Z = X*2 # new variable Z created with local scope
   W = X+5 # use module W as instructed above

   if Z > W:
      # pow is a 'builtin-scope' name
      print( pow(Z,W) )
      return Z
   else:
      return Y # no local Y so uses module version

print("W,Y = ", W, Y )
for n in [2,4,6]:
```

```
   print( "Spam(%d) returned: " % n, spam(n) )
   print( "W,Y = ", W, Y )
```

VBScript

VBScript takes a fairly straightforward approach to scoping rules: if a variable is outside a function or subroutine then it is globally visible, if a variable is inside a function or subroutine it is local to that module. The programmer is responsible for managing all naming conflicts that might arise. Because all VBScript variables are created using the `Dim` statement there is never any ambiguity about which variable is meant as is the case with Python.

There are some slight twists that are unique to web pages, namely that regardless of `<script>` tag boundaries global variables are visible across an entire file, not just within the `<script>` tag in which they are defined.

We will illustrate those points in the following code:

```
<script type="text/vbscript">
Dim aVariable
Dim another
aVariable = "This is global in scope"
another = "A Global can be visible from a function"
</script>

<script type="text/vbscript">
Sub aSubroutine
   Dim aVariable
   aVariable = "Defined within a subroutine"
   MsgBox aVariable
   MsgBox another
End Sub
</script>

<script type="text/vbscript">
MsgBox aVariable
aSubroutine
MsgBox aVariable
</script>
```

There are a couple of extra scoping features in VBScript that allow you to make variables accessible across files on a web page (e.g from an index frame to a content frame and vice-versa). However we won't be going into that level of web page programming here so I'll simply alert you to the existence of the `Public` and `Private` keywords.

And JavaScript too

JavaScript follows much the same rules, variables declared inside a function are only visible within the function. Variables outside a function can be seen inside the function as well as by code on the outside. As with VBScript there are no conflicts as to which variable is intended because variables are explicitly created with the `var` statement.

Here is the equivalent example as above but written in JavaScript:

```
<script type="text/javascript">
var aVariable, another;  // global variables
```

```
aVariable = "This is Global in scope<BR>";
another = "A global variable can be seen inside a function<BR>";

function aSubroutine(){
   var aVariable;        // local variable
   aVariable = "Defined within a function<BR>";
   document.write(aVariable);
   document.write(another);
}

document.write(aVariable);
aSubroutine();
document.write(aVariable);

</script>
```

This should, by now, be straightforward.

Things to Remember

- Scoping and Namespaces are different terms for the same thing.
- The concepts are the same in every language but the precise rules can vary.
- Python has 5 scopes - class, nested, file (global), function (local) and built-in. The last 3 are the most significant in everyday programming.
- VBScript and JavaScript each have 2 scopes - file (global) and function (local).

Previous Next Contents

If you have any questions or feedback on this page send me mail at: alan.gauld@btinternet.com

Regular Expressions

What will we cover?
• What regular expressions are • How to use regular expressions in Python programs • Regex support in JavaScript and VBScript

Definition

Regular expressions are groups of characters that describe a larger group of characters. They describe a *pattern* of characters for which we can search in a body of text. They are very similar to the concept of wild cards used in file naming on most operating systems, whereby an asterisk(*) can be used to represent any sequence of characters in a file name. So `*.py` means any file ending in `.py`. In fact filename wild-cards are a very small subset of regular expressions.

Regular expressions are extremely powerful tools and most modern programming languages either have built in support for using regular expressions or have libraries or modules available that you can use to search for and replace text based on regular expressions. A full description of them is outside the scope of this tutor, indeed there is at least one whole book dedicated to regular expressions and if your interest is roused I recommend that you investigate the O'Reilly book.

One interesting feature of regular expressions is that they manifest similarities of structure to programs. Regular expressions are patterns constructed from smaller units. These units are:

- single characters
- wildcard characters
- character ranges or sets and
- groups which are surrounded by parentheses.

Note that because groups are a unit, so you can have groups of groups and so on to an arbitrary level of complexity. We can combine these units in ways reminiscent of a programming language using sequences, repetitions or conditional operators. We'll look at each of these in turn. So that we can try out the examples you will need to import the `re` module and use it's methods. For convenience I will assume you have already imported `re` in most of the examples shown.

Sequences

As ever, the simplest construct is a sequence and the simplest regular expression is just a sequence of characters:

```
red
```

This will match, or find, any occurrence of the three letters `'r','e'` and `'d'` in order, in a string. Thus the words **red**, lette**red** and c**red**ible would all be found because they contain 'red' within them. To provide greater control over the outcome of matches we can supply some special characters (known as *metacharacters*) to limit the scope of the search:

Metacharacters used in sequences

Expression	Meaning	Example
^red	only at the start of a line	**red** ribbons are good

red$	only at the end of a line	I love **red**
\Wred	only at the start of a word	it's **red**irected by post
red\W	only at the end of a word	you cove**red** it already

The metacharacters above are known as *anchors* because they fix the position of the regular expression within a sentence or word. There are several other anchors defined in the `re` module documentation which we don't cover in this topic.

Sequences can also contain wildcard characters that can substitute for any character. The wildcard character is a period. Try this:

```
>>> import re
>>> re.match('be.t', 'best')
<_sre.SRE_Match object at 0x01365AA0>
>>> re.match('be.t', 'bess')
```

The message in angle brackets tells us that the regular expression `'be.t'`, passed as the first argument matches the string `'best'` passed as the second argument. `'be.t'` will also match 'beat', 'bent', 'belt', etc. The second example did not match because `'bess'` didn't end in `t`, so no MatchObject was created. Try out a few more matches to see how this works. (Note that `match()` only matches at the front of a string, not in the middle, we can use `search()` for that, as we shall see later!)

The next unit is a *range* or *set*. This consists of a collection of letters enclosed in square brackets and the regular expression will search for any one of the enclosed letters.

```
>>> re.match('s[pwl]am', 'spam')
<_sre.SRE_Match object at 0x01365AD8>
```

This would also match 'swam' or 'slam' but not 'sham' since 'h' is not included in the regular expression set.

By putting a `^` sign as the first element of the group we can say that it should look for any character *except* those listed, thus in this example:

```
>>> re.match('[^f]ool', 'cool')
<_sre.SRE_Match object at 0x01365AA0>
>>> re.match('[^f]ool','fool')
```

we can match 'cool' and 'pool' but we will not match 'fool' since we are looking for any character except 'f' at the beginning of the pattern.

Finally we can group sequences of characters, or other units, together by enclosing them in parentheses, which is not particularly useful in isolation but is useful when combined with the repetition and conditional features we look at next.

Repetition

We can also create regular expressions which match repeated sequences of characters by using some more special characters. We can look for a repetition of a single character or group of characters using the following metacharacters:

Metacharacters used in repetition

Expression	Meaning	Example
'?'	zero or one of the preceding character. Note the zero part there since that can trip you up if you aren't careful.	pythonl?y matches: **pythony** **pythonly**
'*'	looks for zero or more of the preceding character.	pythonl*y matches both of the above, plus: **pythonlly** **pythonllly** etc.
'+'	looks for one or more of the preceding character.	pythonl+y matches: **pythonly** **pythonlly** **pythonllly** etc.
{n,m}	looks for n to m repetitions of the preceding character.	fo{1,2} matches: **fo** or **foo**

All of these repetition characters can be applied to groups of characters too. Thus:

```
>>> re.match('(.an){1,2}s', 'cans')
<_sre.SRE_Match object at 0x013667E0>
```

The same pattern will also match: 'cancans' or 'pans' or 'canpans' but not 'bananas' since there is no character before the second 'an' group. (How could we modify the search to work with bananas as well? Hint: Look at the other repeat specifiers - and don't forget the extra 'a' at the end of bananas)

There is one caveat with the `{m,n}` form of repetition which is that it does not limit the match to only `n` units. Thus the example in the table above, `fo{1,2}` will successfully match `fooo` because it matches the `foo` at the beginning of `fooo`. Thus if you want to limit how many characters are matched you need to follow the multiplying expression with an anchor or a negated range. In our case `fo{1,2}[^o]` would prevent `fooo` from matching since it says match 1 or 2 'o's followed by anything other than an 'o' - but, it must be followed by something, so now 'foo' doesn't match! This illustrates the fickle nature of regular expressions. They can be very difficult to get just right and you need to be very careful to test them thoroughly! The actual pattern needed to allow 'foo', and 'foobar' but not 'fooo' is: `'fo{1,2}[^o]*$'`. That is, 'fo' or 'foo' followed by zero or more non o's and the end of the line. (In fact even this is not completely foolproof, but we need to cover a few more elements before we can really nail it!)

Greedy expressions

Regular expressions are said to be greedy. What that means is that the matching and searching functions will match as much as possible of the string rather than stopping at the first complete match. Normally this doesn't matter too much but when you combine wildcards with repetition operators you can wind up grabbing more than you expect.

Consider the following example. If we have a regular expression like `a.*b` that says we want to find an `a` followed by any number of characters up to a `b` then the match function will search from the first `a` to the last `b`. That is to say that if the searched string includes more than one 'b' all but the last one will be included in the `.*` part of the expression. Thus in this example:

```
re.match('a.*b',’abracadabra')
```

The MatchObject has matched all of `abracadab`. Not just the first `ab`. This greedy matching behaviour is one of the most common errors made by new users of regular expressions.

To prevent this ‘greedy’ behaviour simply add a ‘?’ after the repition character, like so:

```
re.match('a.*?b','abracadabra')
```

which will now only match ‘ab’.

Conditionals

The final piece in the jigsaw is to make the regular expression search for optional elements or to select one of several patterns. We’ll look at each of these options separately:

Optional elements

You can specify that a character is optional using the zero or more repetition metacharacters:

```
>>> re.match('computer?d?', 'computer')
<re.MatchObject instance at 864890>
```

will match `compute`, `computer` or `computed`. However, it will also match `computerd`, which we don’t want.

By using a range within the expression we can be more specific. Thus:

```
>>> re.match('compute[rd]$','computer')
<re.MatchObject instance at 874390>
```

will select only `computer` and `computed` but reject the unwanted `computerd`.

And if we add a `?` after the range we can also allow `compute` to be selected but still avoid `computerd`.

Optional Expressions

In addition to matching options from a list of characters we can also match based on a choice of sub-expressions. We mentioned earlier that we could group sequences of characters in parentheses, but in fact we can group any arbitrary regular expression in parentheses and treat it as a unit. In describing the syntax I will use the notation `(RE)` to indicate any such regular expression grouping.

The situation we want to examine here is the case whereby we want to match a regular expression containing `(RE)xxxx` or `(RE)yyyy` where `xxxx` and `yyyy` are different patterns. Thus, for example we want to match both `premature` and `preventative`. We can do this by using a selection metacharacter (`|`):

```
>>> regexp = 'pre(mature|ventative)'
>>> re.match(regexp,'premature')
<re.MatchObject instance at 864890>
>>> re.match(regexp,'preventative')
<re.MatchObject instance at 864890>
```

```
>>> re.match(regexp,'prelude')
```

Notice that when defining the regular expression we had to include the full text of both options inside the parentheses, rather than just `(e|v)` otherwise the option would have been restricted to `prematureentative` or `prematurventative`. In other words only the letters `e` and `v` would have formed the options not the full length groups.

Now, using this technique we can come back to the example above where we want to capture 'fo' or 'foo' but not 'fooo' plus whatever comes after. We left it with a regular expression consisting of: `fo{1,2}[^o]*$`. The problem with this one is that if the string following the 'fo' or 'foo' contains an 'o' the match fails. By using a choice of expressions we can get round that. We want the match to work where our pattern is either the end of the line or followed by any non 'o' character. That looks like: `fo{1,2}($|[^o])`. And that finally gives us what we wanted. Remember, when using regular expressions, always test thoroughly to ensure you are not catching more than you want, and are catching all that you want.

A Few Extras

The `re` module has many features which we don't discuss here so it is worth studying the module documentation. One area I'd like to draw to your attention is the set of *flags* that you can use when compiling expressions with the re.compile() function. These flags control things like whether the pattern matches across lines, or ignores case etc.

Another feature that you can find in the standard Python distribution is a regular expression testing tool. It allows you to type in an expression then try different values against it to see if they match. You can find this in the `Tools/Scripts` folder and the file is `redemo.py`. Unfortunately there is a small bug in the version that ships with Python v3.1. The import statement at the top needs to be changed from

```
from TKinter import *
```

to

```
from tkinter import *
```

If you make that small change it should work fine, and by the time you read this it should have been fixed in the distribution too. Have fun!

Using Regular Expressions in Python.

We've seen a little of what regular expressions look like but what can we do with them? And how do we do it in Python? To take the first point first, we can use them as very powerful search tools in text. We can look for lots of different variations of text strings in a single operation, we can even search for non printable characters such as blank lines using some of the metacharacters available. We can also replace these patterns using the methods and functions of the re module. We've already seen the `match()` function at work, there are several other functions, some of which are described below:

re Module functions and methods

Function/Method	Effect
match(RE,string)	if RE matches the *start* of the string it returns a match object

search(RE,string)	if RE is found *anywhere* within the string a match object is returned
split(RE, string)	like string.split() but uses the RE as a separator
sub(RE, replace, string)	returns a string produced by substituting replace for re at the first matching occurrence of RE. Note this function has several additional features, see the documentation for details.
findall(RE, string)	Finds all occurrences of RE in string, returning a list of match objects
compile(RE)	produces a regular expression object which can be reused for multiple operations with the same RE. The object has all of the above methods but with an implied re and is more efficient than using the function versions.

Note that this is not a full list of re's methods and functions and that those listed have some optional parameters that can extend their use. The listed functions are the most commonly used operations and are sufficient for most needs.

A Practical Example Using Regular Expressions

As an example of how we might use regular expressions in Python let's create a program that will search an HTML file for an IMG tag that has no ALT section. If we find one we will add a message to the owner to create more user friendly HTML in future!

```
import re

# detect 'IMG' in upper/lower case allowing for
# zero or more spaces between the < and the 'I'
img = '< *[iI][mM][gG] '

# allow any character up to the 'ALT' or 'alt' before >
alt = img + '.*[aA][lL][tT].*>'

# open file and read it into list
filename = input('Enter a filename to search ')
inf = open(filename,'r')
lines = inf.readlines()

# if the line has an IMG tag and no ALT inside
# add our message as an HTML comment
for i in range(len(lines)):
  if ( re.search(img,lines[i]) and not
       re.search(alt,lines[i]) ):
        lines[i] += '<!-- PROVIDE ALT TAGS ON IMAGES! -->\n'

# Now write the altered file and tidy up.
inf.close()
outf = open(filename,'w')
outf.writelines(lines)
outf.close()
```

Notice two points about the above code. First we use re.search instead of re.match because `search` finds the patterns anywhere in the string whereas `match` only looks at the start of the string. Secondly we put an outer pair of parentheses around the two tests. These are not strictly necessary but they allow us to break the test into two lines which are easier to read, especially if there are many expressions to be combined.

This code is far from perfect because it doesn't consider the case where the IMG tag may be split over several lines, but it illustrates the technique well enough for our purposes. Of course such wanton vandalism of HTML files shouldn't really be encouraged, but then again anyone who doesn't provide ALT tags probably deserves all they get!

Finally, regular expressions have limitations and for formally defined data structures, like HTML, there are often other tools, known as *parsers* that are more effective, reliable, and easier to use correctly, than regular expressions. But for complex searches in free text regular expressions can solve a lot of problems. Just be sure to test thoroughly.

We'll see regular expressions at work again in the Grammar Counter case study, meantime experiment with them and check out the other methods in the re module. We really have just scratched the surface of what's possible using these powerful text processing tools.

JavaScript

JavaScript has good support for regular expressions built into the language. In fact the string search operations we used earlier are actually regular expression searches, we simply used the most basic form - a simple sequence of characters. All of the rules we discussed for Python apply equally to Javascript except that regular expressions are surrounded in slashes(/) instead of quotes. Here are some examples to illustrate their use:

```
<Script type="text/javascript">
var str = "A lovely bunch of bananas";
document.write(str + "<BR>");
if (str.match(/^A/)) {
   document.write("Found string beginning with A<BR>");
}
if (str.match(/b[au]/)) {
   document.write("Found substring with either ba or bu<BR>");
}
if (!str.match(/zzz/)) {
   document.write("Didn't find substring zzz!<BR>");
}
</Script>
```

The first two succeed the third doesn't, hence the negative test.

VBScript

VBScript does not have built in regular expressions like JavaScript but it does have a Regular Expression object that can be instantiated and used for searches, replacement etc. It can also be controlled to ignore case and to search for all instances or just one. It is used like this:

```
<Script type="text/vbscript">
Dim regex, matches
Set regex = New RegExp

regex.Global = True
regex.Pattern = "b[au]"

Set matches = regex.Execute("A lovely bunch of bananas")
If matches.Count > 0 Then
   MsgBox "Found " & matches.Count & " substrings"
End If
</Script>
```

That's all I'll cover here but there is a wealth of subtle sophistication in regular expressions, we have literally just touched on their power in this short topic. Fortunately there is also a wealth of online information about their use, plus the excellent O'Reilly book mentioned at the start. My advice is to take it slowly and get accustomed to their vagaries as well as their virtues.

Points to remember
• Regular expressions are text patterns which can improve the power and efficiency of text searches • Regular expressions are notoriously difficult to get right and can lead to obscure bugs - handle with care. • Regular Expressions are not a cure all and often a more sophisticated approach may be needed, if it doesn't work after say 3 attempts consider another approach!

Previous Next Contents

If you have any questions or feedback on this page send me mail at: alan.gauld@btinternet.com

Object Oriented Programming

What will we cover?

- What is an object?
- What is a Class?
- What are polymorphism and inheritance?
- Creating, Storing and using objects

What is Object Oriented Programming?

Now we move onto what might have been termed an advanced topic up until about 10 years ago. Nowadays '*Object Oriented Programming* has become the norm. Languages like Java and Python embody the concept so much that you can do very little without coming across objects somewhere. So what's it all about?

The best introductions are, in my opinion:

- *Object Oriented Analysis* by Peter Coad & Ed Yourdon.
- *Object Oriented Analysis and Design with Applications* by Grady Booch (the 1st edition if you can find it)
- *Object Oriented Software Construction* by Bertrand Meyer (definitely the 2nd edition of this one)

These increase in depth, size and academic exactitude as you go down the list. For most non-professional programmers' purposes the first is adequate. For a more programming focused introduction try *Object Oriented Programming* by Timothy Budd(2nd edition). This uses several languages to illustrate object oriented programming techniques. It is much more strongly oriented towards writing programs than any of the other books which cover the whole gamut of theory and principle behind object orientation, at the design level as well as at the code level. Finally for a whole heap of info on all topics OO try the Web link site at: http://www.cetus-links.org

Assuming you don't have the time nor the inclination to research all these books and links right now, I'll give you a brief overview of the concept. (**Note:**Some people find OO hard to grasp others 'get it' right away. Don't worry if you come under the former category, you can still use objects even without really 'seeing the light'.)

One final point: it is possible to implement an Object Oriented design in a non OO language through coding conventions, but it's usually an option of last resort rather than a recommended strategy. If your problem fits well with OO techniques then it's best to use an OO language. Most modern languages, including Python, VBScript and JavaScript support OOP quite well. That having been said I will be using Python throughout all the examples and only showing the basic concepts in VBScript and JavaScript with little additional explanation.

Data and Function - together

Objects are collections of data and functions that operate on that data. These are bound together so that you can pass an object from one part of your program and they automatically get access to not only the data *attributes* but the *operations* that are available too. This combining of data and function is the very essence of Object Oriented Programming and is known as *encapsulation*. (Some

programming languages make the data invisible to users of the object and thus require that the data be accessed via the object's methods. This technique is properly known as *data hiding*, however in some texts data hiding and encapsulation are used interchangeably.)

As an example of encapsulation, a string object would store the character string but also provide *methods* to operate on that string - search, change case, calculate length etc.

Objects use a *message passing* metaphor whereby one object passes a message to another object and the receiving object responds by executing one of its operations, a *method*. So a method is *invoked* on receipt of the corresponding message by the owning object. There are various notations used to represent this but the most common mimics the access to items in modules - a dot. Thus, for a fictitious widget class:

```
w = Widget() # create new instance, w, of widget
w.paint()  # send the message 'paint' to it
```

This would cause the paint method of the widget object to be invoked.

Defining Classes

Just as data has various types so objects can have different types. These collections of objects with identical characteristics are collectively known as a *class*. We can define classes and create *instances* of them, which are the actual objects. We can store references to these objects in variables in our programs.

Let's look at a concrete example to see if we can explain it better. We will create a message class that contains a string - the message text - and a method to print the message.

```
class Message:
    def __init__(self, aString):
        self.text = aString
    def printIt(self):
        print( self.text )
```

Note 1: One of the methods of this class is called `__init__` and it is a special method called a *constructor*. The reason for the name is that it is called when a new object instance is created or constructed. Any variables assigned (and hence created in Python) inside this method will be unique to the new instance. There are a number of special methods like this in Python, nearly all distinguished by the `__xxx__` naming format. The exact timing of when a constructor is called varies between languages, in Python `init` gets called after the instance has actually been created in memory, in other languages the constructor actually returns the instance itself. The difference is sufficiently subtle that you don't usually need to worry about it.

Note 2: Both the methods defined have a first parameter `self`. The name is a convention but it indicates the object instance. As we will soon see this parameter is filled in by the interpreter at run-time, not by the programmer. Thus `printIt` is called, on an instance of the class (see below), with no arguments: `m.printIt()`.

Note 3: We called the class `Message` with a capital 'M'. This is purely convention, but it is fairly widely used, not just in Python but in other OO languages too. A related convention says that method names should begin with a lowercase letter and subsequent words in the name begin with uppercase letters. Thus a method called "calculate current balance" would be written: `calculateCurrentBalance`.

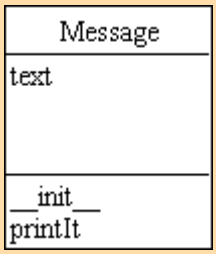
You may want to briefly revisit the 'Raw Materials' section and look again at 'user defined types'. The Python address example should be a little clearer now. Essentially the only kind of user-defined type in Python is a class. A class with attributes but no methods (except `__init__`) is effectively equivalent to a construct called a `record` or `struct` in some programming languages..

A Graphical Notation

The software engineering community have adopted a graphical notation for describing classes and objects and their relationships to each other. This notation is called the *Unified Modelling Language* (or UML) and is a powerful design tool. In total UML contains many diagrams and icons but we will only look at a few here that may help you grasp the concepts.

The first and most important icon we meet in UML is the class description, it consists of a box with three compartments. The top compartment contains the class name, the middle compartment contains the class attributes, or data, and the bottom compartment contains the methods, or functions, of the class.

The Message class defined above would look like this:



We will show other UML icons as we develop the topic and introduce new concepts supported by the notation.

Using Classes

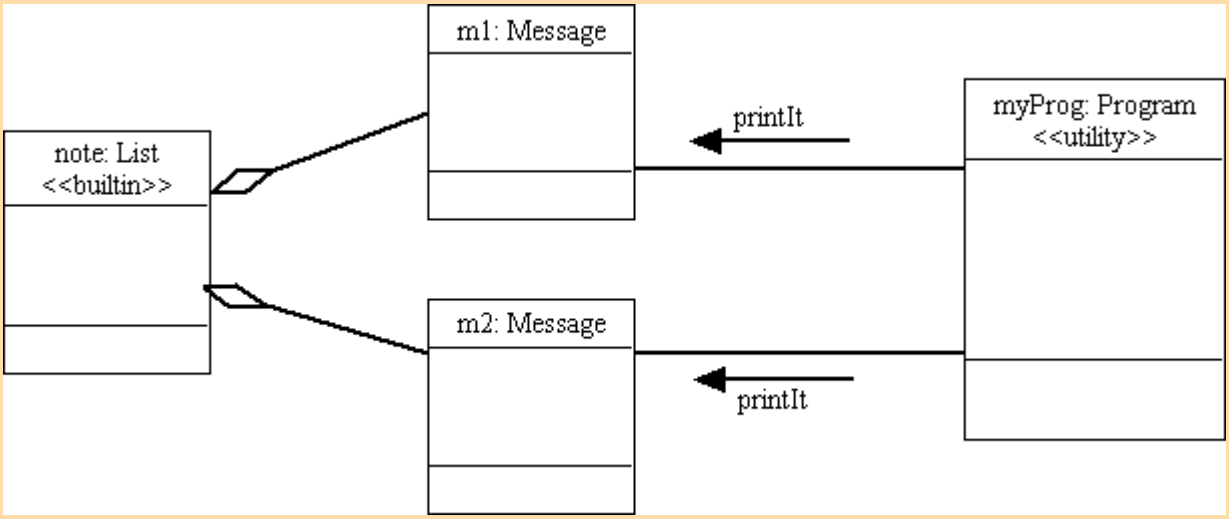
Having defined a class we can now create instances of our Message class and manipulate them:

```
m1 = Message("Hello world")
m2 = Message("So long, it was short but sweet")

note = [m1, m2] # put the objects in a list
for msg in note:
    msg.printIt() # print each message in turn
```

So in essence you just treat the class as if it was a standard Python data type, which was after all the purpose of the exercise!

UML also has an icon for an object or instance. It is the same as the class icon, except we usually leave the bottom two boxes blank. The name is made up of the object or instance name followed by the class name with a colon in between. Thus `m1:Message` tells us that `m1` is an instance of the `Message` class.

Our message example would be drawn like this:



Note that the `List` class represents the normal Python list type (as indicated by the word `builtin` being within angle brackets, a construct known as a *stereotype* in UML). The lines with diamonds indicate that the list *contains* the `Message` objects. The `MyProg` object likewise is stereotyped as being a *utility* class, which means, in this case, that it does not exist as a class within the program but is a product of the environment. (Operating system facilities are often shown this way, as are libraries of functions.) The solid lines from `myProg` to `Message` indicate that the `myProg` "object" has an *association* with, or reference to, the `Message` objects. The arrows adjacent to these lines indicate that the `myProg` "object" sends the `printIt` message to each of the `Message` objects. In effect object messages are transmitted via associations.

What is "self"?

No, it's not a philosophical debate, it's one of the questions most often asked by new Python OOP programmers. Every method definition in a class in Python starts with a parameter called *self*. The actual name `self` is just a convention, but like many programming conventions consistency is good so let's stick with it! (As you'll see later JavaScript has a similar concept but uses the name `this` instead.)

So what is `self` all about? Why do we need it?

Basically self is just a reference to the current instance. When you create an instance of the class the instance contains its own data (as created by the constructor) but not of the methods. Thus when we send a message to an instance and it calls the corresponding method, it does so via an internal reference to the class. It passes a reference to itself (self!) to the method so that the class code knows which instance to use.

Let's look at a relatively familiar example. Consider a GUI application which has lots of Button objects. When a user presses a button the method associated with a button press is activated - but how does the Button method know which of the buttons has been pressed? The answer is by referring to the self value which will be a reference to the actual button instance that was pressed. We'll see this in practice when we get to the GUI topic a little later.

So what happens when a message is sent to an object? It works like this:

- the client code calls the instance (sending the message in OOP speak).
- The instance calls the class method, passing a reference to itself (self).
- The class method then uses the passed reference to pick up the instance data for the receiving object.

You can see this in action in this code sequence, notice that we can explicitly call the class method, as we do in the last line:

```
>>> class C:
...   def __init__(self, val): self.val = val
...   def f(self): print "hello, my value is:", self.val
...
>>> # create two instances
>>> a = C(27)
>>> b = C(42)
>>> # first try sending messages to the instances
>>> a.f()
hello, my value is 27
>>> b.f()
hello, my value is 42
>>> # now call the method explicitly via the class
>>> C.f(a)
hello, my value is 27
```

So you see we can call the methods via the instance, in which case Python fills in the self parameter for us, or explicitly via the class, in which case we need to pass the self value explicitly.

Now you might be wondering why, if Python can provide the invisible reference between the instance and its class can't Python also magically fill in the self by itself? The answer is that Guido van Rossum designed it this way! Many OOP languages do indeed hide the self parameter, but one of the guiding principles of Python is that "explicit is better than implicit". You soon get used to it and after a while not doing it seems strange.

Same thing, Different thing

What we have so far is the ability to define our own types (classes) and create instances of these and assign them to variables. We can then pass messages to these objects which trigger the methods we have defined. But there's one last element to this OO stuff, and in many ways it's the most important aspect of all.

If we have two objects of different classes but which support the same set of messages but with their own corresponding methods then we can collect these objects together and treat them identically in our program but the objects will behave differently. This ability to behave differently to the same input messages is known as *polymorphism*.

Typically this could be used to get a number of different graphics objects to draw themselves on receipt of a 'paint' message. A circle draws a very different shape from a triangle but provided they both have a paint method we, as programmers, can ignore the difference and just think of them as 'shapes'.

Let's look at an example, where instead of drawing shapes we calculate their areas:

First we create Square and Circle classes:

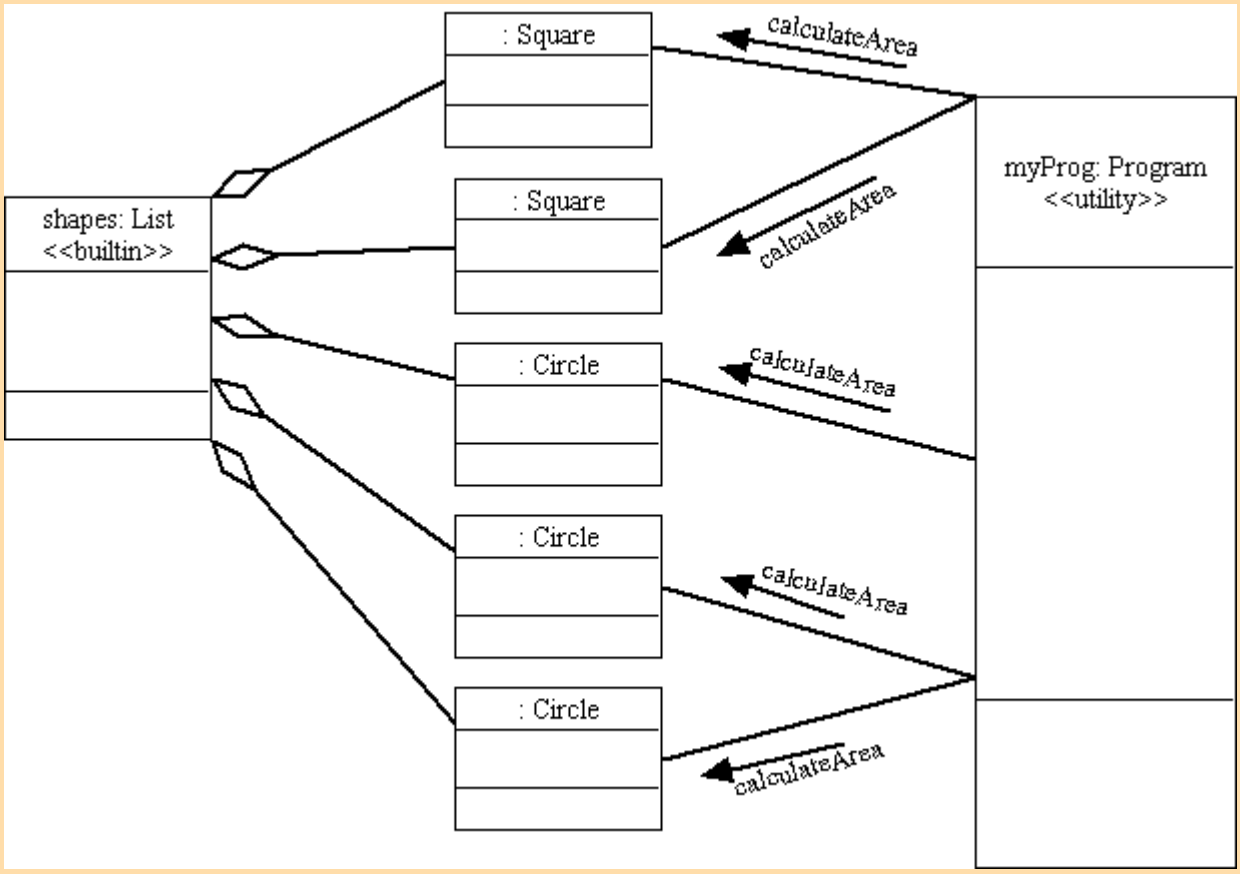
```
class Square:
    def __init__(self, side):
        self.side = side
    def calculateArea(self):
        return self.side**2
```

```
class Circle:
    def __init__(self, radius):
        self.radius = radius
    def calculateArea(self):
        import math
        return math.pi*(self.radius**2)
```

Now we can create a list of shapes (either circles or squares) and then print out their areas:

```
shapes = [Circle(5),Circle(7),Square(9),Circle(3),Square(12)]

for item in shapes:
    print "The area is: ", item.calculateArea()
```

Now if we combine these ideas with modules we get a very powerful mechanism for reusing code. Put the class definitions in a module - say 'shapes.py' and then simply import that module when we want to manipulate shapes. This is exactly what has been done with many of the standard Python modules, which is why accessing methods of an object looks a lot like using functions in a module.



Here we see a more complex object diagram. Notice that in this case the objects within the list do not have names because we did not explicitly create variables for them. In this case we just show a blank before the colon and class name. However, the diagram is starting to get very busy. For this reason we only draw object diagrams when necessary to illustrate some unusual feature of the design. Instead we use more sophisticated features of class diagrams to show the relationships, as we'll see in the examples below.

Inheritance

Inheritance is often used as a mechanism to implement polymorphism. Indeed in many OO languages it is the only way to implement polymorphism. It works as follows:

A class can *inherit* both attributes and operations from a *parent* or *super* class. This means that a new class which is identical to another class in most respects does not need to re-implement all the methods of the existing class, rather it can inherit those capabilities and then *override* those that it wants to do differently (like the `calculateArea` method in the case above)

Again an example might illustrate this best. We will use a *class hierarchy* of bank accounts where we can deposit cash, obtain the balance and make a withdrawal. Some of the accounts provide interest (which, for our purposes, we'll assume is calculated on every deposit - an interesting innovation to the banking world!) and others charge fees for withdrawals.

The BankAccount class

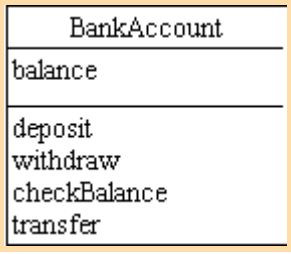
Let's see how that might look. First let's consider the attributes and operations of a bank account at the most general (or *abstract*) level.

It's usually best to consider the operations first then provide attributes as needed to support these operations. So for a bank account we can:

- *Deposit* cash,
- *Withdraw* cash,
- *Check current balance* and
- *Transfer* funds to another account.

To support these operations we will need a bank account ID (for the transfer operation) and the current balance. In this example we will just use the variable to which we assign the object, but in a more general case we would likely create an ID attribute which stored a unique reference. We will also need to store the balance.

In UML that would look like:



We can now create a class to support that:

```
# first create a custom exception class
class BalanceError(Exception):
      value = "Sorry you only have $%6.2f in your account"

class BankAccount:
    def __init__(self, initialAmount):
       self.balance = initialAmount
       print( "Account created with balance %5.2f" % self.balance )

    def deposit(self, amount):
       self.balance = self.balance + amount

    def withdraw(self, amount):
       if self.balance >= amount:
          self.balance = self.balance - amount
       else:
          raise BalanceError( BalanceError.value % self.balance )

    def checkBalance(self):
       return self.balance
```

```
    def transfer(self, amount, account):
       try:
          self.withdraw(amount)
          account.deposit(amount)
       except BalanceError:
          print( BalanceError.value % self.balance )
```

Note 1: We check the balance before withdrawing and also use an exception to handle errors. Of course there is no Python error type `BalanceError` so we needed to create one of our own - it's simply an subclass of the standard `Exception` class with a string value. The string `value` is defined as an attribute of the exception class purely as a convenience, it ensures that we can generate standard error messages every time we raise an error. When we `raise BalanceError` we pass the internal format string `value` filled in with the current value of the object's `balance`. Notice that we didn't use `self` when defining the value in `BalanceError`, that's because `value` is a shared attribute across all instances, it is defined at the class level and known as a *class variable*. We access it by using the class name followed by a dot: `BalanceError.value` as seen above. Now, when the error generates its traceback it concludes by printing out the formatted error string showing the current balance.

Note 2: The `transfer` method uses the `BankAccount`'s `withdraw/deposit` *member functions* or methods to do the transfer. This is very common in OO and is known as *self messaging*. It means that *derived classes* can implement their own versions of `deposit/withdraw` but the `transfer` method can remain the same for all account types.

OK, now that we have defined our `BankAccount` as a *base class* we can get back to inheritance which is what we are supposed to be discussing! Let's look at our first *sub class*>.

The InterestAccount class

Now we use inheritance to provide an account that adds interest (we'll assume a default of 3%) on every deposit. It will be identical to the standard BankAccount class except for the deposit method and the initialisation of the interest rate. So we simply override those:

```
class InterestAccount(BankAccount):
   def __init__(self, initialAmount, interest=0.03):
       BankAccount.__init__(self, initialAmount)
       self.interest = interest
   def deposit(self, amount):
       BankAccount.deposit(self,amount)
       self.balance = self.balance * (1 + self.interest)
```

Note that we call the `BankAccount` initialisation method at the beginning of `__init__()` which takes care of all the initialisation of the super class for us, we just need to initialise the new `interest` attribute that we introduced here. Because we call it via the class (see the discussion of "self" above) we need to explicitly include `self` as an argument.

And that's it. We begin to see the power of OOP, all the other methods have been inherited from `BankAccount` (by putting `BankAccount` inside the parentheses after the new class name). Notice that once again `deposit` called the *superclass*'s `deposit` method rather than copying the code. Now if we modify the `BankAccount` deposit to include some kind of error checking the *sub-class* will gain those changes automatically.

The ChargingAccount class

This account is again identical to a standard `BankAccount` class except that this time it charges a default fee of $3 for every withdrawal. As for the `InterestAccount` we can create a class inheriting from `BankAccount` and modifying the `init` and `withdraw` methods.

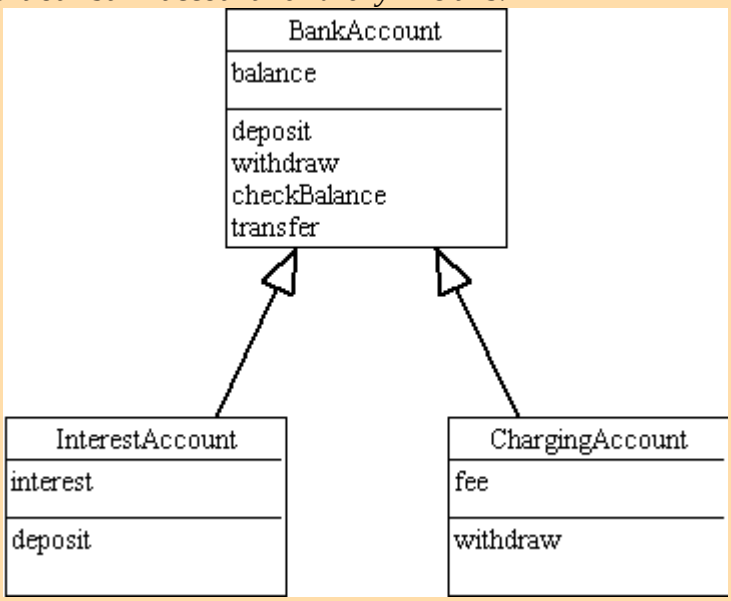
```
class ChargingAccount(BankAccount):
    def __init__(self, initialAmount, fee=3):
        BankAccount.__init__(self, initialAmount)
        self.fee = fee

    def withdraw(self, amount):
        BankAccount.withdraw(self, amount+self.fee)
```

Note 1: We store the fee as an *instance variable* so that we can change it later if necessary. Notice that we again call the inherited `__init__` just like any other method.

Note 2: We simply add the fee to the requested withdrawal and call the `BankAccount.withdraw` method to do the real work.

Note 3: We introduce a side effect here in that a charge is automatically levied on transfers too, but that's probably what we want, so is OK.

In UML we represent inheritance with a solid arrow pointing from the sub class to the superclass. We can now represent our bank account *heirarchy* like this:



Notice we only list the methods and attributes that have changed or been added in sub classes.

Testing our system

To check that it all works try executing the following piece of code (either at the Python prompt or by creating a separate test file).

```
from bankaccount import *

# First a standard BankAccount
a = BankAccount(500)
b = BankAccount(200)
a.withdraw(100)
# a.withdraw(1000)
```

```
a.transfer(100,b)
print( "A = ", a.checkBalance() )
print( "B = ", b.checkBalance() )


# Now an InterestAccount
c = InterestAccount(1000)
c.deposit(100)
print( "C = ", c.checkBalance() )


# Then a ChargingAccount
d = ChargingAccount(300)
d.deposit(200)
print( "D = ", d.checkBalance() )
d.withdraw(50)
print( "D = ", d.checkBalance() )
d.transfer(100,a)
print( "A = ", a.checkBalance() )
print( "D = ", d.checkBalance() )


# Finally transfer from charging account to the interest one
# The charging one should charge and the interest one add
# interest
print( "C = ", c.checkBalance() )
print( "D = ", d.checkBalance() )
d.transfer(20,c)
print( "C = ", c.checkBalance() )
print( "D = ", d.checkBalance() )
```

Now uncomment the line `a.withdraw(1000)` to see the exception at work.

That's it. A reasonably straightforward example but it shows how inheritance can be used to quickly extend a basic framework with powerful new features.

We've seen how we can build up the example in stages and how we can put together a test program to check it works. Our tests were not complete in that we didn't cover every case and there are more checks we could have included - like what to do if an account is created with a negative amount...

Test Driven Development

Many professional programmers use a technique known as Test Driven Development (TDD) in which they write their tests before they write their code. This may initially sound bizarre but it allows them to test their code repeatedly as they develop it and move from a state where every test fails to one where every test passes. At that point their program should work properly!

So popular is this that special tools have been developed to assist with this approach. Python has several such tools including the unittest module that is in the standard library. TDD is a good approach when writing serious code however in the context of a tutorial it would simply hide the main code that we are trying to study amongst a mass of test cases so I won't be using it here. But its a concept you might like to investigate once you start writing longer programs.

Collections of Objects

One problem that might have occurred to you is how we deal with lots of objects. Or how to manage objects which we create at runtime. It's all very well creating bank accounts statically as we did above:

```
acc1 = BankAccount(...)
acc2 = BankAccount(...)
acc3 = BankAccount(...)
etc...
```

But in the real world we don't know in advance how many accounts we need to create. How do we deal with this? Lets consider the problem in more detail:

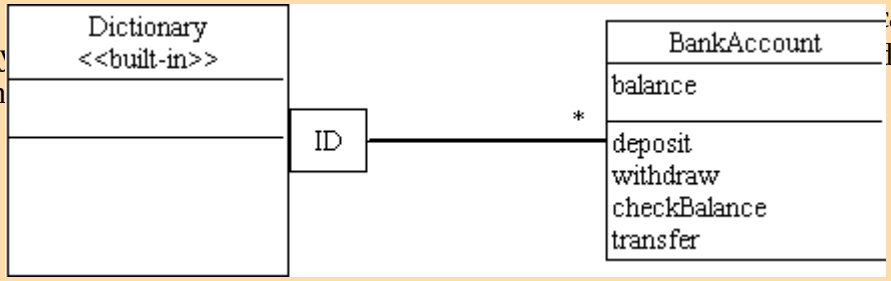
We need some kind of 'database' that allows us to find a given bank account by its owners name (or more likely their bank account number - since one person can have many accounts and several persons can have the same name...)

Finding something in a collection given a unique key....hmmm, sounds like a dictionary! Let's see how we'd use a Python dictionary to hold dynamically created objects:

```
from bankaccount import BankAccount
import time

# Create new function to generate unique id numbers
def getNextID():
    ok = input("Create account[y/n]? ")
    if ok[0] in 'yY':  # check valid input
       id = time.time() # use current time as basis of ID
       id = int(id) % 10000 # convert to int and shorten to 4 digits
    else: id = -1  # which will stop the loop
    return id

# Let's create some accounts and store them in a dictionary
accountData = {}  # new dictionary
while True:          # loop forever
   id = getNextID()
   if id == -1:
      break       # break forces an exit from the while loop
   bal = float(input("Opening Balance? "))  # convert string to float
   accountData[id] = BankAccount(bal) # use id to create new dictionary entry
   print( "New account created, Number: %04d, Balance %0.2f" % (id, bal) )

# Now let's access the accounts
for id in accountData.keys():
    print( "%04d\t%0.2f" % (id, accountData[id].checkBalance()) )

# and find a particular one
# Enter non-number to force exception and end program
while True:
   id = int(input("Which account number? "))
   if id in accountData.keys():
      print( "Balance = %0.2d" % accountData[id].checkBalance() )
   else: print( "Invalid ID" )
```

Of course the key you use for the dictionary can be anything that uniquely identifies the object, it could be one of its attributes, like `balance` say (except that balance would not be a very good unique key!). Anything at all that is unique. You might find it worthwhile going back to the raw materials chapter and reading the dictionary section again, they really are very useful containers.

We can represent that graphically in UML using a class diagram. The dictionary is shown as a class which has a *relationship* with many BankAccounts. This is shown by the asterisk on the line connecting the classes. An asterisk is used because that is the symbol used in regular expressions to indicate zero or [illegible] can be shown in a number of ways [illegible] because of their richness an[illegible]



Notice the use of a stereotype on the `Dictonary` to show it is a built-in class. Notice also the box attached to the association showing that the key is the `ID` value. If we had been using a simple list we would not have had the box and the line would have directly connected the two classes. This use of class relationships and cardinality is how we avoid the need for very large complex Object diagrams. We can focus on the abstract relationships between classes rather than the myriad of physical relationships between individual instances.

Saving Your Objects

One snag with all of this is that you lose your data when the program ends. You need some way of saving objects too. As you get more advanced you will learn how to use databases to do that but we will look at using a simple text file to save and retrieve objects. (If you are using Python there are a couple of modules called Pickle and Shelve) that do this much more effectively but as usual I'll try to show you the generic way to do it that will work in any language. Incidentally the technical term for the ability to save and restore objects is *Persistence*.

The generic way to do this is to create `save` and `restore` methods at the highest level object and override in each class, such that they call the inherited version and then add their locally defined attributes:

```
class A:
   def __init__(self,x,y):
     self.x = x
     self.y = y

   def save(self,fn):
     f = open(fn,"w")
     f.write(str(self.x)+ '\n') # convert to a string and add newline
     f.write(str(self.y)+'\n')
     return f             # for child objects to use

   def restore(self, fn):
     f = open(fn)
     self.x = int(f.readline()) # convert back to original type
     self.y = int(f.readline())
     return f

class B(A):
   def __init__(self,x,y,z):
     A.__init__(self,x,y)
     self.z = z

   def save(self,fn):
     f = A.save(self,fn)  # call parent save
     f.write(str(self.z)+'\n')
```

```
      return f         # in case further children exist

   def restore(self, fn):
      f = A.restore(self,fn)
      self.z = int(f.readline())
      return f

# create instances
a = A(1,2)
b = B(3,4,5)

# save the instances
a.save('a.txt').close() # remember to close the file
b.save('b.txt').close()

# retrieve instances
newA = A(5,6)
newA.restore('a.txt').close() # remember to close the file
newB = B(7,8,9)
newB.restore('b.txt').close()
print( "A: ",newA.x,newA.y )
print( "B: ",newB.x,newB.y,newB.z )
```

Note: The values printed out are the restored values not the ones we used to create the instances.

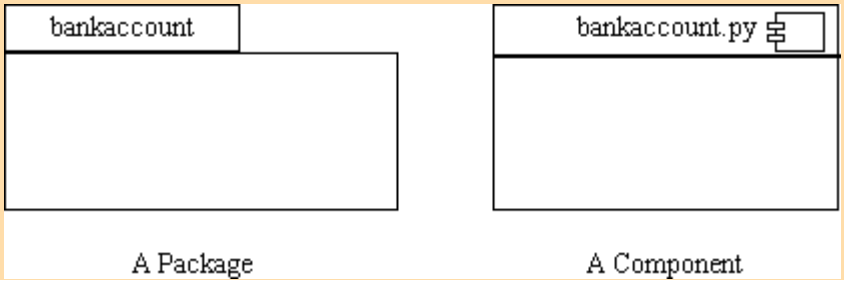
The key thing is to override the save/restore methods in each class and to call the parent method as the first step. Then in the child class only deal with child class attributes. Obviously how you turn an attribute into a string and save it is up to you the programmer but it must be output on a single line. When restoring you simply reverse the storing process.

One big snag with this approach is that you need to create a separate file for each object. In a real world example that could mean thousands of very small files. This quickly gets cumbersome and so using a database to store the objects becomes necessary. We will look at how to do that in a later topic, but the basic principles remain the same.

Mixing Classes and Modules

Modules and classes both provide mechanisms for controlling the complexity of a program. It seems reasonable that as programs get bigger we would want to combine these features by putting classes into modules. Some authorities recommend putting each class into a separate module but I find this simply creates an explosion of modules and increases rather than decreases complexity. Instead I group classes together and put the group into a module. Thus in our example above I might put all the bank account class definitions in one module, `bankaccount`, say, and then create a separate module for the application code that uses the module.

We can represent that graphically in UML in two ways. The logical grouping of the classes can be represented using a *Package* or we can represent the physical file as a *component*. The icons for these are shown below:



The intention is that the Package icon should look somewhat like a folder in a typical file explorer tool. The little icon at top right in the component icon is actually the old UML symbol for a component but this was a bit cumbersome in diagrams when trying to draw lines showing relationships between components so they demoted it to a small embellishment in UML 2.0

That's all the UML I'll be covering, if you find it interesting and a useful way to visualise your design then a Google search will throw up lots of references and tutorials and you will find some UML drawing tools too, although the shapes are sufficiently easy to draw that you can use just about any vector graphics package.

A simplified representation of that design would be:

```
# File: bankaccount.py
#
# Implements a set of bank account classes
####################

class BankAccount: ....

class InterestAccount: ...

class ChargingAccount: ...
```

And then to use it:

```
import bankaccount

newAccount = bankaccount.BankAccount(50)
newChrgAcct = bankaccount.ChargingAccount(200)


# now do stuff
```

But what happens when we have two classes in different modules that need to access each others details? The simplest way is to import both modules, create local instances of the classes we need and pass the instances of one class to the other instance's methods. Passing whole objects around is what makes it object *oriented* programming. You don't need to extract the attributes out of one object and pass them into another, just pass the entire object. Now if the receiving object uses a polymorphic message to get at the information it needs then the method will work with any kind of object that supports the message.

Let's make that more concrete by looking at an example. Let's create a short module called `logger` that contains two classes. The first class, called `Logger`, logs activity in a file. This logger will have a single method `log()` which takes a "loggable object" as a parameter. The other class in our module is a `Loggable` class that can be inherited by other classes to work with the logger. It looks like this:

```
# File: logger.py
#
# Create Loggable and Logger classes for logging activities
# of objects
```

```
############

class Loggable:
   def activity(self):
       return "This needs to be overridden locally"

class Logger:
   def __init__(self, logfilename = "logger.dat"):
       self._log = open(logfilename,"a")

   def log(self, loggedObj):
       self._log.write(loggedObj.activity() + '\n')

   def __del__(self):
       self._log.close()
```

Note that we have provided a *destructor* method (`__del__`) to close the file when the logger object is deleted or garbage collected. This is another "magic method" in Python (as shown by the double '_' characters) similar in many ways to `__init__()`

Also notice that we've called the log attribute `_log` with a '_' character in front of the name. This is another common naming convention in Python, like using capitalized words for class names. A single underscore indicates that the attribute is not intended to be accessed directly, but only via the methods of the class.

Now before we can use our module we will create a new module which defines loggable versions of our bank account classes:

```
# File: loggablebankaccount.py
#
# Extend Bank account classes to work with logger module.
################################

import bankaccount, logger

class LoggableBankAccount(bankaccount.BankAccount, logger.Loggable):
    def activity(self):
       return "Account balance = %d" % self.checkBalance()

class LoggableInterestAccount(bankaccount.InterestAccount,
                              logger.Loggable):
    def activity(self):
       return "Account balance = %d" % self.checkBalance()

class LoggableChargingAccount(bankaccount.ChargingAccount,
                              logger.Loggable):
    def activity(self):
       return "Account balance = %d" % self.checkBalance()
```

Notice we are using a feature called *multiple inheritance*, where we inherit not one but two parent classes. This isn't strictly needed in Python since we could just have added an `activity()` method to our original classes and achieved the same effect but in statically typed OOP languages such as Java or C++ this technique would be necessary so I will show you the technique here for future reference.

The sharp eyed amongst you may have noticed that the `activity()` method in all three classes is identical. That means we could save ourselves some typing by creating an intermediate type of loggable account class that inherits Loggable and only has an activity method. We can then create our three different loggable account types by inheriting from that new class as well as from the vanilla Loggable class. Like this:

```
class LoggableAccount(logger.Loggable):
     def activity(self):
         return "Account balance = %d" % self.checkBalance()

class LoggableBankAccount(bankaccount.BankAccount, LoggableAccount):
     pass

class LoggableInterestAccount(bankaccount.InterestAccount, LoggableAccount):
     pass

class LoggableChargingAccount(bankaccount.ChargingAccount, LoggableAccount):
     pass
```

It doesn't save a lot of code but it does mean we only have one method definition to test and maintain instead of three identical methods. This type of programming, where we introduce a superclass with shared functionality is sometimes called *mixin programming* and the minimal class is called a *mixin class*. It is a common outcome of this style that the final class definitions have little or no body but a long list of inherited classes, just as we see here. It's also quite common that mixin classes do not themselves inherit from anything, although in this case we did. In essence it's just a way of adding a common method (or set of methods) to a class or set of classes via the power of inheritance. (The term mixin originates in the world of ice cream parlours where different flavours of ice cream are added (or mixed in) to vanilla to produce a new flavour. The first language to support this style was called Flavors which was a popular dialect of Lisp for a while.)

Now we come to the point of this exercise which is to show our application code creating a logger object and some bank accounts and passing the accounts to the logger, even though they are all defined in different modules!

```
# Test logging and loggable bank accounts.
##############

import logger
import loggablebankaccount as lba

log = logger.Logger()

ba = lba.LoggableBankAccount(100)
ba.deposit(700)
log.log(ba)

intacc = lba.LoggableInterestAccount(200)
intacc.deposit(500)
log.log(intacc)
```

Note the use of the `as` keyword to create a shortcut name when importing `loggablebankaccount`

Note also that once we have created the local instances we no longer need to use the module prefix and because there is no direct access from one object to the other, it is all via messages, there is no need for the two class definition modules to directly refer to each other either. Finally notice also that the Logger works with instances of both `LoggableBankAccount` and `LoggableInterestAccount` because they both support the `Loggable` *interface*. Compatibility of object interfaces via polymorphism is the foundation upon which all OOP programs are built.

I should point out that a much more sophisticated logging system is included in the standard library `logging` module, this one was purely to demonstrate some techniques. If you want logging facilities in your own programmes investigate the standard `logging` module first of all.

Hopefully this has given you a taste of Object Oriented Programming and you can move on to some of the other online tutorials, or read one of the books mentioned at the beginning for more information and examples. Now we will briefly look at how OOP is done in VBScript and JavaScript.

OOP in VBScript

VBScript supports the concept of objects and allows us to define classes and create instances, however it does *not* support the concepts of inheritance or polymorphism. VBScript is therefore what is known as *Object Based* rather than fully Object Oriented. Nonetheless the concepts of combining data and function in a single object remain useful, and a limited form of inheritance is possible using a technique called delegation which we discuss below.

Defining classes

A class is defined in VBScript using the `Class` statement, like this:

```
<script type=text/VBScript>
Class MyClass
   Private anAttribute
   Public Sub aMethodWithNoReturnValue()
       MsgBox "MyClass.aMethodWithNoReturnValue"
   End Sub
   Public Function aMethodWithReturnValue()
       MsgBox "MyClass.aMethodWithReturnValue"
       aMethodWithReturnValue = 42
   End Function
End Class
</script>
```

This defines a new class called `MyClass` with an attribute called `anAttribute` which is only visible to the methods inside the class, as indicated by the keyword `Private`. It is conventional to declare data attributes to be `Private` and most methods to be `Public`. This is known as data hiding and has the advantage of allowing us to control access to the data by forcing methods to be used and the methods can do data quality checks on the values being passed in and out of the object. Python provides its own mechanism for achieving this but it is beyond the scope of this tutorial.

Creating Instances

We create instances in VBScript with a combination of the `Set` and `New` keywords. The variable to which the new instance is assigned must also have been declared with the `Dim` keyword as is the usual VBScript style.

```
<script type=text/VBScript>
```

```
Dim anInstance
Set anInstance = New MyClass
</script>
```

This creates an instance of the class declared in the previous section and assigns it to the `anInstance` variable.

Sending Messages

Messages are sent to instances using the same *dot notation* used by Python.

```
<script type=text/VBScript>
Dim aValue
anInstance.aMethodWithNoReturnValue()
aValue = anInstance.aMethodWithReturnValue()
MsgBox "aValue = " & aValue
</script>
```

The two methods declared in the class definition are called, in the first case there is no return value, in the second we assign the return to the variable `aValue`. There is nothing unusual here apart from the fact that the subroutine and function are preceded by the instance name.

Inheritance and Polymorphism

VBScript as a language does not provide any inheritance mechanism nor any mechanism for polymorphism. However we can fake it to some degree by using a technique called *delegation*. This simply means that we define an attribute of the sub class to be an instance of the theoretical parent class. We then define a method for all of the "inherited" methods which simply calls (or delegates to), in turn, the method of the parent instance. Let's subclass `MyClass` as defined above:

```
<script type=text/VBScript>
Class SubClass
   Private parent
   Private Sub Class_Initialize()
      Set parent = New MyClass
   End Sub
   Public Sub aMethodWithNoReturnValue()
      parent.aMethodWithNoREturnVAlue
   End Sub
   Public Function aMethodWithReturnValue()
      aMethodWithReturnValue = parent.aMethodWithReturnValue
   End Function
   Public Sub aNewMethod
      MsgBox "This is unique to the sub class"
   End Sub
End Class

Dim inst,aValue
Set inst = New SubClass
inst.aMethodWithNoReturnVAlue
aValue = inst.aMethodWithReturnValue
inst.aNewMethod
MsgBox "aValue = " & CStr(aValue)
</script>
```

The key points to note here are the use of the private attribute `parent` and the special, private method `Class_Initialise`. The former is the superclass delegate attribute and the latter is the equivalent of Pythons `__init__` method for initializing instances when they are created, it is the VBScript constructor in other words.

OOP in JavaScript

JavaScript supports objects using a technique called *prototyping*. This means that there is no explicit class construct in JavaScript and instead we can define a class in terms of a set of functions or a dictionary like concept known as an *initializer*.

Defining classes

The most common way to define a JavaScript "class" is to create a function with the same name as the class, effectively this is the constructor, but is not contained within any other construct. It looks like this:

```
<script type=text/JavaScript>
function MyClass(theAttribute)
{
   this.anAttribute = theAttribute;
};
</script>
```

You might notice the keyword `this` which is used in the same way as Python's `self` as a placeholder reference to the current instance.

We can add new attributes to the class later using the built in `prototype` attribute like this:

```
<script type=text/JavaScript>
MyClass.prototype.newAttribute = null;
</script>
```

This defines a new attribute of `MyClass` called `newAttribute`.

Methods are added by defining a normal function then assigning the function name to a new attribute with the name of the method. Normally the method and function have the same name, but there is nothing to stop you calling the methods something different, as illustrated below:

```
<script type=text/JavaScript>
function oneMethod(){
    return this.anAttribute;
}
MyClass.prototype.getAttribute = oneMethod;
function printIt(){
    document.write(this.anAttribute + "<BR>");
};
MyClass.prototype.printIt = printIt;
</script>
```

Of course it would be more convenient to define the functions first then finish up with the constructor and assign the methods inside the constructor and this is in fact the normal approach, so that the full class definition looks like this:

```
<script type=text/JavaScript>
function oneMethod(){
    return this.anAttribute;
};

function printIt(){
    document.write(this.anAttribute + "<BR>");
};

function MyClass(theAttribute)
{
   this.anAttribute = theAttribute;
   this.getAttribute = oneMethod;
   this.printIt = printIt;
};
</script>
```

Creating Instances

We create instances of classes using the keyword `new`, like this:

```
<script type=text/JavaScript>
var anInstance = new MyClass(42);
</script>
```

Which creates a new instance called `anInstance`.

Sending Messages

Sending messages in JavaScript is no different to our other languages, we use the familiar dot notation.

```
<script type=text/JavaScript>
document.write("The attribute of anInstance is: <BR>");
anInstance.printIt();
</script>
```

Inheritance and Polymorphism

Unlike VBScript it is possible to use JavaScript's prototyping mechanism to inherit from another class. It is rather more complex than the Python technique but is not completely unmanageable, but it is, in my experience, a relatively uncommon technique among JavaScript programmers.

The key to inheritance in JavaScript is the `prototype` keyword (we used it in passing in the code above). By using `prototype` we can effectively add features to an object after it has been defined. We can see this in action here:

```
<script type="text/javascript">
function Message(text){
   this.text = text;
   this.say = function(){
        document.write(this.text + '<br>');
        };
};

msg1 = new Message('This is the first');
msg1.say();
```

```
Message.prototype.shout = function(){
    alert(this.text);
    };

msg2 = new Message('This gets the new feature');
msg2.shout();

/* But so did msg1...*/
msg1.shout();

</script>
```

Note 1: We added the new `alert` method using `prototype` after creating instance `msg1` of the class but the feature was available to the existing instance as well as to the instance, `msg2` created after the addition. That is, the new feature gets added to all instances of `Message` both existing and new.

Note 2: We used function in a new way here. It effectively is used to create a function object which is assigned to the object property. That is:

```
obj.func = function(){...};
```

is equivalent to saying:

```
function f(){....};
obj.func = f;
```

We will see a similar concept in Python when we get to the Functional Programming topic.

This prototyping feature gives rise to the interesting capability to change the behavior of built-in JavaScript objects, either adding new features or changing the way existing features function! Use this capability with great care if you don't want to spend your time grappling with really confusing bugs.

This use of `prototype` as a mechanism for adding functionality to existing classes has the disadvantage that it alters the existing instance behaviors and changes the original class definition.

More conventional style inheritance is available too, as shown below:

```
<script type="text/javascript">
function Parent(){
   this.name = 'Parent';
   this.basemethod = function(){
       alert('This is the parent');
       };
};

function Child(){
   this.parent = Parent;
   this.parent();
   this.submethod = function(){
       alert('This from the child');
       };
};

var aParent = new Parent();
var aChild = new Child();

aParent.basemethod();
```

```
aChild.submethod();
aChild.basemethod();

</script>
```

The key point to note here is that the `Child` object has access to the `basemethod` without it being explicitly granted, it has inherited it from the parent class by virtue of the assignment/call pair of lines:

```
   this.parent = Parent;
   this.parent();
```

within the `Child` class definition. And thus we have inherited the `basemethod` from the `Parent` class!

We can, of course, use the same delegation trick we used with VBScript. Here is the VBScript example translated into JavaScript:

```
<script type=text/JavaScript>
function noReturn(){
   this.parent.printIt();
};

function returnValue(){
      return this.parent.getAttribute();
};

function newMethod(){
      document.write("This is unique to the sub class<BR>");
};

function SubClass(){
   this.parent = new MyClass(27);
   this.aMethodWithNoReturnValue = noReturn;
   this.aMethodWithReturnValue = returnValue;
   this.aNewMethod = newMethod;
};

var inst, aValue;
inst = new SubClass(); // define superclass
document.write("The sub class value is:<BR>");
inst.aMethodWithNoReturnValue();
aValue = inst.aMethodWithReturnValue();
inst.aNewMethod();
document.write("aValue = " + aValue);
</script>
```

We will see classes and objects being used in the following topics and case studies. It is not always obvious to a beginner how this, apparently complex, construct can make programs easier to write and understand but hopefully as you see classes being used in real programs it will become clearer. One thing I would like to say is that, for very small programs they do not really help and almost certainly will make the program longer. However as your programs start to get bigger - over about 100 lines say - then you will find that classes and objects can help to keep things organized and even reduce the amount of code you write.

If you are one of those who finds the whole OOP concept confusing don't panic, many people have programmed for their whole lives without ever creating a single class! On the other hand, if you can get to grips with objects it does open up some powerful new techniques.

Things to Remember

- Classes *encapsulate* data and function into a single entity.
- Classes are like cookie cutters, used to create *instances*, or *objects*.
- Objects communicate by sending each other *messages*.
- When an object receives a message it executes a corresponding *method*.
- Methods are functions stored as attributes of the class.
- Classes can *inherit* methods and data from other classes. This makes it easy to extend the capabilities of a class without changing the original.
- *Polymorphism* is the ability to send the same message to several different types of object and each behaves in its own particular way in response.
- Encapsulation, Polymorphism and Inheritance are all properties of *Object Oriented* programming languages.
- VBScript and JavaScript are called *Object Based* languages because while they support encapsulation, they do not fully support inheritance and polymorphism.

Previous Next Contents

If you have any questions or feedback on this page send me mail at: alan.gauld@btinternet.com

Event Driven Programming

What will we cover?

- How does an event driven program differ from a batch program?
- How to write an event loop
- How to use an event framework such as Tkinter

So far we have been looking at batch oriented programs. Recall that programs can be *batch oriented*, whereby they start, do something then stop, or *event driven* where they start, wait for *events* and only stop when told to do so - by an event. How do we create an event driven program? We'll look at this in two ways - first we will simulate an event environment then we'll create a very simple GUI program that uses the operating system and environment to generate events.

Simulating an Event Loop

Every event driven program has a loop somewhere that catches received events and processes them. The events may be generated by the operating environment, as happens with virtually all GUI programs or the program itself may go looking for events as is often the case in embedded control systems such as used in cameras etc.

We will create a program that looks for precisely one type of event - keyboard input - and processes the results until some quit event is received. In our case the quit event will be the space key. We will process the incoming events in a very simple manner - we will simply print the ASCII code for that key. We'll use Python for this because it has a nice, easy to use function for reading keys one at a time - `getch()`. This function comes in two varieties depending on the operating system you use. If you are using Linux it's found in the `curses` module, if you use Windows it's in the `msvcrt` module. I'll use the Windows version first then I'll discuss the Linux option in more detail. You will need to run these programs from an OS command prompt since any IDE, like IDLE, being a GUI, will capture the keystrokes differently.

First we implement the event handler functions that will be called when a keypress is detected then the main program body which simply starts up the event gathering loop and calls the appropriate event handling function when a valid event is detected.

```
import msvcrt

def doKeyEvent(key):
    if key == '\x00' or key == '\xe0': # non ASCII
       key = msvcrt.getch() # fetch second character
    print( ord(key), end='')

def doQuitEvent(key):
    raise SystemExit


# First, clear the screen of clutter then warn the user
# of what to do to quit
lines = 25 # set to number of lines in console
for line in range(lines): print()

print( "Hit space to end..." )
print()

# Now mainloop runs "forever"
```

```
while True:
   ky = msvcrt.getch()
   length = len(ky)
   if length != 0:
      # send events to event handling functions
      if ky == " ": # check for quit event
         doQuitEvent(ky)
      else:
         doKeyEvent(ky)
```

Notice that what we do with the events is of no interest to the main body, it simply collects the events and passes them to the event handlers. This independence of event capture and processing is a key feature of event driven programming.

Note: Where the key was non ASCII - a Function key for example - we needed to fetch a second character from the keyboard, this is because these special keys actually generate pairs of bytes and `getch` only retrieves one at a time. The actual value of interest is the second byte.

Linux and MacOS X programmers can't use the `msvcrt` library so must use another module called `curses` instead. The resultant code is very similar to the windows code but there are a few modifications required, as shown below:

```
import curses as c

def doKeyEvent(key):
    if key == '\x00' or key == '\xe0': # non ASCII key
       key = screen.getch()     # fetch second character
    screen.addstr(str(key)+' ') # uses global screen variable

def doQuitEvent(key):
    raise SystemExit

# clear the screen of clutter, stop characters auto
# echoing to screen and then tell user what to do to quit
screen = c.initscr()
c.noecho()
screen.addstr("Hit space to end...\n")

# Now mainloop runs "forever"
while True:
     ky = screen.getch()
     if ky != -1:
       # send events to event handling functions
       if ky == ord(" "): # check for quit event
         doQuitEvent(ky)
       else:
         doKeyEvent(ky)

c.endwin()
```

You'll see that the usual print commands don't work under curses and instead we have to use curses own screen handling functions. Further, curses `getch` returns `-1` when there is no key pressed rather than an empty string. Aside from that the logic of the program is identical to the Windows version.

Note that the `curses.endwin()` should restore your screen to normal but in some cases it may not work completely. If you wind up with an invisible cursor, no carriage return etc. You can fix it if you exit Python with Ctrl-D and use the Linux command:

```
$ stty echo -nl
```

Hopefully that will restore things to normal.

Note: At the time of writing I can't actually test the Linux code under Python v3 since I don't have access to a Linux distribution with Python v3 installed. This code is copied from the Python v2 tutorial. Hopefully it works as-is. If not you might have to do some experimentation to get it working. If that's the case please let me know. I'll remove this caveat once I get the chance to test the code properly.

If we were creating this as a framework for use in lots of projects we would probably include a call to an *initialization* function at the start and a *cleanup* function at the end. The programmer could then use the loop part and provide his own initialization, processing and cleanup functions.

That's exactly what most GUI type environments do, in that the loop part is embedded in the operating environment or framework and applications are *contractually required* to provide the event handling functions and *hook* these into the event loop in some way.

Let's see that in action as we explore Python's Tkinter GUI library.

A GUI program

For this exercise we'll use the Python Tkinter toolkit. This is a Python wrapper around the Tk toolkit originally written as an extension to Tcl and also available for Perl and Ruby. The Python version is an object oriented framework which is, in my opinion, considerably easier to work with than the original Tk version. We will look much more closely at the principles of GUI programming in the GUI topic.

I am not going to dwell much on the GUI aspects in this topic, rather I want to focus on the *style* of programming - using Tkinter to handle the event loop and leaving the programmer to create the initial GUI and then process the events as they arrive.

In the example we create an application class `KeysApp` which creates the GUI in the `__init__` method and *binds* the space key to the `doQuitEvent` method. The class also defines the required `doQuitEvent` method.

The GUI itself simply consists of a text entry *widget* whose default behavior is to echo characters typed onto the display.

Creating an application class is quite common in OO event driven environments because there is a lot of synergy between the concepts of events being sent to a program and messages being sent to an object. The two concepts map on to each other very easily. An event handling function thus becomes a method of the application class.

Having defined the class we simply create an instance of it and then send it the `mainloop` message.

The code looks like this:

```
# Use from X import * to save having to preface everything
# as tkinter.xxx
from tkinter import *
import sys
```

```
# Create the application class which defines the GUI
# and the event handling methods
class KeysApp(Frame):
    def __init__(self): # use constructor to build GUI
        Frame.__init__(self)
        self.txtBox = Text(self)
        self.txtBox.bind("<space>", self.doQuitEvent)
        self.txtBox.pack()
        self.pack()

    def doQuitEvent(self,event):
        sys.exit()


# Now create an instance and start the event loop running
myApp = KeysApp()
myApp.mainloop()
```

Note: If you run this from inside IDLE you will find the program doesn't close properly but simply prints an exit message in the shell window. Don't worry, that's just IDLE trying to be helpful. If you run it from a command prompt everything should be just fine.

Notice that we don't even implement a key event handler! That's because the default behavior of the Text widget is to print out the keys pressed. However that does mean our programs are not really functionally equivalent. In the console version we printed the ASCII codes of all keys rather than only printing the alphanumeric versions of printable keys as we do here. There's nothing to prevent us capturing all of the keypresses and doing the same thing. To do so we would add the following line to the `__init__` method:

```
self.txtBox.bind("<Key>", self.doKeyEvent)
```

And the following method to process the event:

```
def doKeyEvent(self,event):
    str = "%d\n" % event.keycode
    self.txtBox.insert(END, str)
    return "break"
```

Note 1: the key value is stored in the `keycode` field of the event. I had to look at the source code of Tkinter.py to find that out... Recall that curiosity is a key attribute of a programmer?!

Note 2: `return "break"` is a magic signal to tell Tkinter not to invoke the default event processing for that widget. Without that line, the text box displays the ASCII code followed by the actual character typed, which is not what we want here.

That's enough on Tkinter for now. This isn't meant to be a Tkinter tutorial, that's the subject of the next topic. There are also several books on using Tk and Tkinter.

Event Driven Programming in VBScript and JavaScript

Both VBScript and JavaScript can be used in an event driven manner when programming a web browser. Normally when a web page containing script code is loaded the script is executed in a batch fashion as the page loads. However if the script contains nothing but function definitions the execution will do nothing but define the functions ready for use, but the functions will not be called

initially. Instead, in the HTML part of the page the functions will be bound to HTML elements - usually within a Form element - such that when events occur the functions are called. We have already seen this in the JavaScript example of getting user input, when we read the input from an HTML form. Let's look at that example again more closely and see how it really is an example of event driven programming within a web page:

```
<script type="text/javascript">
function myProgram(){
   alert("We got a value of " + document.entry.data.value);
}
</script>

<form name='entry'>
<P>Type value then click outside the field with your mouse</P>
<Input Type='text' Name='data' onChange='myProgram()'>
</form>
```

The script part simply defines a JavaScript function, and the definition is executed when the page loads. The HTML code then creates a `Form` with an `Input` element. As part of the `Input` definition we bind the `onChange` event to a short block of JavaScript which simply executes our `myProgram()` event handler. Now when the user changes the content of the `Input` box the browser executes our event handler. The event loop is embedded inside the browser.

VBScript can be used in exactly the same way except that the function definitions are all in VBScript instead of JavaScript, like this:

```
<script type="text/vbscript">
Sub myProgram()
   MsgBox "We got a value of " & Document.entry2.data.value
End Sub
</script>

<form name='entry2'>
<P>Type value then click outside the field with your mouse</P>
<Input Type='text' Name='data' onChange='myProgram()'>
</form>
```

Thus we can see that web browser code can be written in batch form or event driven form or a combination of styles to suit our needs.

Things to remember

- Event loops do not care about the events they detect
- Event handlers handle one single event at a time
- Frameworks such as Tkinter provide an event loop and often some default event handlers too.
- Web browsers provide for both batch and event driven coding styles, or even a mixture of both.

Previous Next Contents

If you have any questions or feedback on this page send me mail at: alan.gauld@btinternet.com

GUI Programming with Tkinter

What will we cover?

- Basic GUI building principles
- Basic widgets
- Simple Tkinter program structure
- GUI and OOP, a perfect match
- wxPython as an alternative to Tkinter

In this topic we look at how a GUI program is assembled in a general sense, then how this is done using Python's native GUI toolkit, Tkinter. This will not be a full blown Tkinter reference nor even a complete tutorial. There is already a very good and detailed tutor linked from the Python web site. This tutorial will instead try to lead you through the basics of GUI programming, introducing some of the basic GUI components and how to use them. We will also look at how Object Oriented programming can help organize a GUI application.

GUI principles

The first thing I want to say is that you won't learn anything new about programming here. Programming a GUI is exactly like any other kind of programming, you can use sequences, loops, branches and modules just as before. What is different is that in programming a GUI you usually use a *Toolkit* and must follow the pattern of program design laid down by the toolkit provider. Each new toolkit will have its own API and set of design rules and you as a programmer need to learn these. This is why most programmers try to standardize on only a few toolkits which are available across multiple languages - learning a new toolkit tends to be much harder than learning a new programming language!

Most windows programming languages come with a toolkit included (usually a thin veneer over the very primitive toolkit built into the windowing system itself). Visual Basic, Delphi and Visual C++/.NET are examples of this.

Java is different in that the language includes its own graphics toolkit (actually more than one!) which runs on any platform that Java runs on - which is almost any platform!

There are other toolkits that you can get separately which can be used on any OS (Unix, Mac, Windows...). These generally have adapters to allow them to be used from many different languages. Some of these are commercial but many are freeware. Examples are: GT/K, Qt, Tk

They all have web sites. For some examples try:

- wxPython, a Python version of the wxWidgets toolkit which is actually written in C++
- PyQt, the Qt toolkit which has "bindings" to most languages.
- pyGTK, the Gimp Toolkit, or GTk which is an opensource project used heavily in the Linux community.

Qt and GT/k are what most Linux applications are written in and are both free for non commercial use (i.e. where you don't sell your programs for profit). Qt can provide a commercial license too if you want to use it for commercial purposes and GTk is licensed under the Gnu GPL which has its own special terms.

The standard Python graphics toolkit (i.e. it comes with the language) is Tkinter which is based on Tk, a fairly old multi-OS toolkit. This is the toolkit we will look at most closely, versions of it are available for Tcl, Haskell, Ruby and Perl as well as Python.

The principles in Tk are slightly different to other toolkits so I will conclude with a very brief look at another popular GUI toolkit for Python (and C/C++) which is more conventional in its approach. But first, some general principles:

As we have already stated several times GUI applications are nearly always event driven in nature. If you don't remember what that means go back and look at the event driven programming topic.

I will assume that you are already familiar with GUIs as a user and will focus on how GUI programs work from a programmer's perspective. I will not be going into details of how to write large complex GUIs with multiple windows, MDI interfaces etc. I will stick to the basics of creating a single window application with some labels, buttons, text boxes and message boxes.

First things first, we need to check our vocabulary. GUI programming has its own set of programming terms. The most common terms are described in the table below:

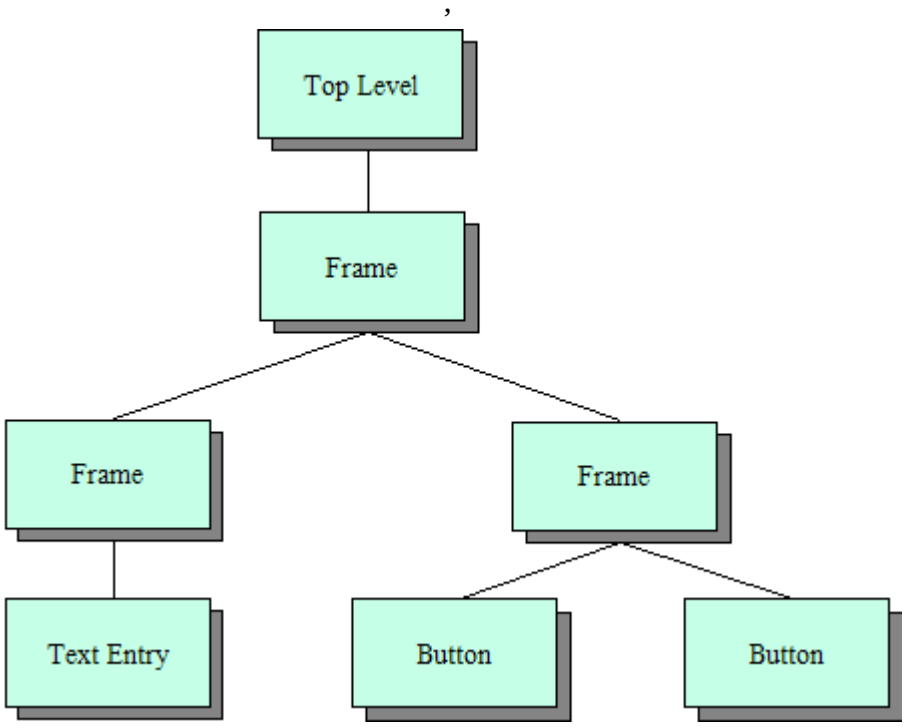
Term	Description
Window	An area of the screen controlled by an application. Windows are usually rectangular but some GUI environments permit other shapes. Windows can contain other windows and frequently every single GUI control is treated as a window in its own right.
Control	A control is a GUI object used for controlling the application. Controls have properties and usually generate events. Normally controls correspond to application level objects and the events are coupled to methods of the corresponding object such that when an event occurs the object executes one of its methods. The GUI environment provides a mechanism for binding events to methods.

Widget	A control, sometimes restricted to visible controls. Some controls (such as timers) can be associated with a given window but are not visible. Widgets are that subset of controls which are visible and can be manipulated by the user or programmer. The widgets that we shall cover are: • Frame • Label • Button • Text Entry • Message box The ones we won't discuss in this topic but are used elsewhere in the tutor are: • Text box • Radio Button Finally, some of the ones not discussed at all are: • Canvas - for drawing • Check button - for multiple selections • Image - for displaying BMP, GIF, JPEG and PNG images • Listbox - for lists! • Menu/MenuButton - for building menus • Scale/Scrollbar - indicating position
Frame	A type of widget used to group other widgets together. Often a Frame is used to represent the complete window and further frames are embedded within it.
Layout	Controls are laid out within a Frame according to a particular set of rules or guidelines. These rules form a Layout. The Layout may be specified in a number of ways, either using on-screen coordinates specified in pixels, using relative position to other components (left, top etc) or using a grid or table arrangement. A coordinate system is easy to understand but difficult to manage when a window is resized etc. Beginners are advised to use non-resizable windows if working with coordinate based layouts.
Child	GUI applications tend to consist of a hierarchy of widgets/controls. The top level Frame comprising the application window will contain sub frames which in turn contain still more frames or controls. These controls can be visualized as a tree structure with each control having a single parent and a number of children. In fact it is normal for this structure to be stored explicitly by the widgets so that the programmer, or more commonly the GUI environment itself, can often perform some common action to a control and all of its children. For example, closing the topmost widget results in all of the child widgets being closed too.

The Containment tree

One very important principle to grasp in GUI programming is the idea of a containment hierarchy. That is, the widgets are contained in a tree like structure with a top level widget controlling the entire interface. It has various child widgets which in turn may have children of their own. Events arrive at a child widget and if it is unable to handle it it will pass the event to its parent and so on up to the top level. Similarly if a command is given to draw a widget it will send the command on down to its children, thus a draw command to the top level widget will redraw the entire application whereas one sent to a button will likely only redraw the button itself.

This concept of events percolating up the tree and commands being pushed down is fundamental to understanding how GUIs operate at the programmer level. It is also the reason that you always need to specify a widget's parent when creating it, so that it knows where it sits in the containment tree. We can draw the containment tree for a simple application that we will create later in this topic like this:



This illustrates the top level widget containing a single `Frame` which represents the outermost window border. This in turn contains two more Frames, the first of which contains a `Text Entry` widget and the second contains the two `Buttons` used to control the application. We will refer back to this diagram later in the topic when we come to build the GUI.

A Tour of Some Common Widgets

In this section we will use the Python interactive prompt to create some simple windows and widgets. Note that because IDLE is itself a Tkinter application you cannot reliably run Tkinter applications within IDLE. You can of course create the files using IDLE as an editor but you must run them from a OS command prompt. Pythonwin users can run Tkinter applications since Pythonwin is built using windows own GUI toolkit, MFC. However even within Pythonwin there are certain unexpected behaviors with Tkinter application. As a result I will use the raw Python prompt from the Operating System.

```
>>> from tkinter import *
```

This is the first requirement of any Tkinter program - import the names of the widgets. You could of course just import the module but it quickly gets tiring typing `tkinter` in front of every component name.

```
>>> top = Tk()
```

This creates the top level widget in our widget hierarchy. All other widgets will be created as children of this.

What happened at this point will depend on where you are typing the program. If, like me, you are using Python from an OS prompt you will have seen that a new blank window has appeared complete with an empty title bar save for a Tk logo as icon and the usual set of control buttons (iconify, maximize etc). If you are using an IDE you may not see anything yet, it will onoly appear when we complete the GUI and start the main event loop running.

We will now add components to this window as we build an application.

```
>>> dir(top)
[....lots of stuff!...]
```

The `dir()` function shows us what names are known to the argument. You can use it on modules but in this case we are looking at the internals of the `top` object, an instance of the `Tk` class. These are the attributes of `top`, and there are a lot of them! Take a look, in particular, for the `children` and `master` attributes which are the links to the widget containment tree. Note also the attribute `_tclCommands`, this is because, as you might recall, Tkinter is built on a Tcl toolkit called Tk.

```
>>> F = Frame(top)
```

Create a *Frame* widget which will in turn contain the child controls/widgets that we use. `Frame` specifies `top` as its first (and in this case only) parameter thus signifying that `F` will be a child widget of `top`.

```
>>> F.pack()
```

Notice that the Tk window (if it's visible) has now shrunk to the size of the added Frame widget - which is currently empty so the window is now very small! The `pack()` method invokes a *Layout Manager* known as the *packer* which is very easy to use for simple layouts but becomes a little clumsy as the layouts get more complex. We will stick with it for now because it is easy to use. Note that widgets will not be visible in our application until we pack them (or use another Layout manager method). We will talk a lot more about Layout Managers later on, after we complete this short program.

```
>>> lHello = Label(F, text="Hello world")
```

Here we create a new object, `lHello`, an instance of the `Label` class, with a parent widget `F` and a `text` attribute of "Hello world". Notice that because Tkinter object constructors tend to have many parameters (each with default values) it is usual to use the named parameter technique of passing arguments to Tkinter objects. Also notice that the object is not yet visible because we haven't packed it yet.

One final point to note is the use of a naming convention: I put a lowercase`l`, for Label, in front of a name, `Hello`, which reminds me of its purpose. Like most naming conventions this is a matter of personal choice, but I find it helps.

```
>>> lHello.pack()
```

Now we can see it. Hopefully yours looks quite a lot like this:

We can specify other properties of the Label such as the font and color using parameters to the object constructor too. We can also access the corresponding properties using the `configure` method of Tkinter widgets, like so:

```
>>> lHello.configure(text="Goodbye")
```

The message changed. That was easy, wasn't it? `configure` is an especially good technique if you need to change multiple properties at once because they can all be passed as arguments. However if you only want to change a single property at a time, as we did above you can treat the object like a dictionary, thus:

```
>>> lHello['text'] = "Hello again"
```

which is shorter and arguably easier to understand.

Labels are pretty boring widgets, they can only display read-only text, albeit in various colors, fonts and sizes. (In fact they can be used to display simple graphics too but we won't bother with that here).

Before we look at another object type there is one more thing to do and that's to set the title of the window. We do that by using a method of the top level widget `top`:

```
>>> F.master.title("Hello")
```

We could have used `top` directly but, as we'll see later, access through the Frame's master property is a useful technique.

```
>>> bQuit = Button(F, text="Quit", command=F.quit)
```

Here we create a new widget - a button. The button has a label "Quit" and is associated with the command `F.quit`. Note that we pass the method name, we do **not** call the method by adding parentheses after it. This means we must pass a function object in Python terms, it can be a built-in method provided by Tkinter, as here, or any other function that we define. The function or method must take no arguments. The `quit` method, like the `pack` method, is defined in a base class and is inherited by all Tkinter widgets, but is usually called at the top window level of the application.

```
>>> bQuit.pack()
```

Once again the `pack` method makes the button visible.

```
>>> top.mainloop()
```

And finally we start the Tkinter event loop. Notice that the Python >>> prompt has now disappeared. That tells us that Tkinter now has control. If you press the `Quit` button the prompt will return, proving that our `command` option worked. Don't expect the window to close, the python interpreter is still running and we only quit the `mainloop` function, the various widgets will be destroyed when Python exits - which in real programs is usually immediately after the mainloop terminates!

Note that if running this from Pythonwin or IDLE you may not have seen anything until this point! And you may get a slightly different result, if so try typing the commands so far into a Python script and running them from an OS command prompt.

In fact it's probably a good time to try that anyhow, after all, it's how most Tkinter programs will be run in practice. Use the principle commands from those we've discussed so far as shown:

```
from tkinter import *

# set up the window itself
top = Tk()
F = Frame(top)
F.pack()

# add the widgets
lHello = Label(F, text="Hello")
lHello.pack()
bQuit = Button(F, text="Quit", command=F.quit)
bQuit.pack()

# set the loop running
top.mainloop()
```

The call to the `top.mainloop` method starts the Tkinter event loop generating events. In this case the only event that we catch will be the button press event which is connected to the `F.quit` method. `F.quit` in turn will terminate the application and this time the window will also close because Python has also exited. Try it, it should look like this:

Notice that I missed the line that changes the window title. Try adding that line in by yourself and check that it works as expected.

Exploring Layout

Note: from now on I'll provide examples as Python script files rather than as commands at the >>> prompt. In most cases I'll only be providing snippets of code so you will have to put in the calls to `Tk()` and the `mainloop()` yourself, use the previous program as a template.

In this section I want to look at how Tkinter positions widgets within a window. We already have seen Frame, Label and Button widgets and those are all we need for this section. In the previous example we used the `pack` method of the widget to locate it within its parent widget. Technically what we are doing is invoking Tk's packer *Layout Manager*. (Another name for Layout Manager is *Geometry Manager*.) The Layout Manager's job is to determine the best layout for the widgets based on hints that the programmer provides, plus constraints such as the size of the window as controlled by the user. Some Layout managers use exact locations within the window, specified in pixels normally, and this is very common in Microsoft Windows environments such as Visual Basic. Tkinter includes a *Placer* Layout Manager which can do this too via a place method. I won't look at that in this tutor because usually one of the other, more intelligent, managers is a better choice, since they take the need to worry about what happens when a window is resized away from us as programmers.

The simplest Layout Manager in Tkinter is the packer which we've been using. The packer, by default, just stacks widgets one on top of the other. That is very rarely what we want for normal widgets, but if we build our applications from Frames then stacking Frames on top of each other is quite a reasonable approach. We can then put our other widgets into the Frames using either the packer or other Layout Manager within each Frame as appropriate. (Each Frame can have its own Layout Manager, but you cannot mix managers within a single frame.) You can see an example of this in action in the Case Study topic.

Even the simple packer provides a multitude of options, however. For example we can arrange our widgets horizontally instead of vertically by providing a `side` argument, like so:

```
lHello.pack(side="left")
bQuit.pack(side="left")
```

That will force the widgets to go to the left thus the first widget (the label) will appear at the extreme left hand side, followed by the next widget (the Button). If you modify the lines in the example above it will look like this:

And if you change the `"left"` to `"right"` then the Label appears on the extreme right and the Button to the left of it, like so:

One thing you notice is that it doesn't look very nice because the widgets are squashed together. The packer also provides us with some parameters to deal with that. The easiest to use is *Padding* and is specified in terms of horizontal padding (`padx`), and vertical padding(`pady`). These values are specified in pixels. Let's try adding some horizontal padding to our example:

```
lHello.pack(side="left", padx=10)
bQuit.pack(side='left', padx=10)
```

It should look like this:

If you try resizing the window you'll see that the widgets retain their positions relative to one another but stay centered in the window. Why is that, if we packed them to the left? The answer is that we packed them into a Frame but the Frame was packed without a side, so it is positioned top, center - the packers default. If you want the widgets to stay at the correct side of the window you will need to pack the Frame to the appropriate side too:

```
F.pack(side='left')
```

Also note that the widgets stay centered if you resize the window vertically - again that's the packers default behavior.

I'll leave you to play with `padx` and `pady` for yourself to see the effect of different values and combinations etc. Between them, `side` and `padx/pady` allow quite a lot of flexibility in the positioning of widgets using the packer. There are several other options, each adding another subtle form of control, please check the Tkinter reference pages for details.

There are a few other layout managers in Tkinter, known as the *grid*, and the *placer*. (In addition the `Tix` module which augments Tkinter provides a `Form` layout manager. We do not cover Tix here.) To use the grid manager you use `grid()` instead of `pack()` and for the placer you call `place()` instead of `pack()`. Each has its own set of options and since I'll only cover the packer in this intro you'll need to look up the Tkinter tutorial and reference for the details. The main points to note are that the grid arranges components in a grid (surprise!) within the window - this can often be useful for dialog boxes with lined up text entry boxes, for example. Many Tkinter users prefer the grid to the packer but for beginners it can take a little getting used to. The placer uses either fixed coordinates in pixels or relative coordinates within a window. The latter allow the component to resize along with the window - always occupying 75% of the vertical space say. This can be useful for intricate window designs but does require a lot of pre-planning - I strongly recommend a pad of squared paper, a pencil and eraser!

Controlling Appearance using Frames and the Packer

The Frame widget actually has a few useful properties that we can use. After all, it's very well having a logical frame around components but sometimes we want something we can see too. This is especially useful for grouped controls like radio buttons or check boxes. The Frame solves this problem by providing, in common with many other Tk widgets, a *relief* property. Relief can have any one of several values: `sunken, raised, groove, ridge` or `flat`. Let's use the `sunken` value on our simple dialog box. Simply change the Frame creation line to:

```
F = Frame(top, relief="sunken", border=1)
```

Note 1: You need to provide a border too. If you don't the Frame will be sunken but with an invisible border - you don't see any difference!

Note 2: that you don't put the border size in quotes. This is one of the confusing aspects of Tk programming is knowing when to use quotes around an option and when to leave them out. In general if it's a numeric or single character value you can leave the quotes off. If it's a mixture of digits and letters or a string then you need the quotes. Likewise with which letter case to use. Unfortunately there is no easy solution, you just learn from experience - Python often gives a list of the valid options in it's error messages!

One other thing to notice is that the Frame doesn't fill the window. We can fix that with another packer option called, unsurprisingly, `fill`. When you pack the frame do it thusly:

```
F.pack(fill="x")
```

This fills horizontally, if you want the frame to fill the entire window just use `fill='y'` too. Because this is quite a common requirement there is a special fill option called `BOTH` so you could type:

```
F.pack(fill="both")
```

The end result of running the script now looks like:

Adding more widgets

Let's now look at a text *Entry* widget. This is the familiar single line of text input box. It shares a lot of the methods of the more sophisticated *Text* widget which we used in the event handling topic and will also use in the case study topic. Essentially we will simply use an Entry to capture what the user types and to clear that text on demand.

Going back to our "Hello World" program we'll add a text Entry widget inside a Frame of its own and then, in a second Frame, put a button that can clear the text that we type into the Entry. We will also add a button to quit the application. This will demonstrate not only how to create and use the Entry widget but also how to define our own event handling functions and connect them to widgets.

```
from tkinter import *

# create the event handler to clear the text
def evClear():
  eHello.delete(0,END)

# create the top level window/frame
top = Tk()
F = Frame(top)
F.pack(fill="both")

# Now the frame with text entry
fEntry = Frame(F, border=1)
eHello = Entry(fEntry)
fEntry.pack(side="top")
eHello.pack(side="left")

# Finally the frame with the buttons.
# We'll sink this one for emphasis
fButtons = Frame(F, relief="sunken", border=1)
bClear = Button(fButtons, text="Clear Text", command=evClear)
bClear.pack(side="left", padx=5, pady=2)
bQuit = Button(fButtons, text="Quit", command=F.quit)
bQuit.pack(side="left", padx=5, pady=2)
fButtons.pack(side="top", fill="x")

# Now run the eventloop
F.mainloop()
```

Note again that we pass the name of the event handlers (`evClear` and `F.quit`), without parentheses, as the `command` argument to the buttons. Note also the use of a naming convention, evXXX to link the event handler with the corresponding widget.

Running the program yields this:

And if you type something in the text entry box then hit the "Clear Text" button it removes it again.

Of course there is not much point in having an Entry widget unless we can get access to the text contained within it. We do this using the `get` method of the widget. I'll illustrate that by copying the text from the widget to a label just before clearing it so that we can always see the last text that the widget held. To do that we need to add a Label widget just under the Entry widget and extend the evClear event handler to copy the text. And just for fun we will colour the label text a light blue. The modified program is shown below with the modifications in bold:

```
from tkinter import *

# create the event handler to clear the text
def evClear():
  lHistory['text'] = eHello.get()
  eHello.delete(0,END)

# create the top level window/frame
top = Tk()
F = Frame(top)
F.pack(fill="both")

# Now the frame with text entry
fEntry = Frame(F, border=1)
eHello = Entry(fEntry)
eHello.pack(side="left")
lHistory = Label(fEntry, foreground="steelblue")
lHistory.pack(side="bottom", fill="x")
fEntry.pack(side="top")

# Finally the frame with the buttons.
# We'll sink this one for emphasis
fButtons = Frame(F, relief="sunken", border=1)
bClear = Button(fButtons, text="Clear Text", command=evClear)
bClear.pack(side="left", padx=5, pady=2)
bQuit = Button(fButtons, text="Quit", command=F.quit)
bQuit.pack(side="left", padx=5, pady=2)
fButtons.pack(side="top", fill="x")

# Now run the eventloop
F.mainloop()
```

Notice that while we have here assigned the text directly to the Label property but we could equally well have assigned it to a normal Python variable for use later in our program.

You might recall that back in the Talking to the User topic we discussed the EasyGUI module and its basic data entry dialog box? You can probably begin to see how such a dialog box can be created. Unfortunately there are a couple of extra bits of information we need before we are ready to do that.

Binding events - from widgets to code

Up till now we have used the command property of buttons to associate Python functions with GUI events. Sometimes we want more explicit control, for example to catch a particular key combination. The way to do that is use the `bind` function to explicitly tie together (or bind) an event and a Python function.

We'll now define a hot key - let's say CTRL-c - to delete the text in the above example. To do that we need to bind the CTRL-C key combination to the same event handler as the Clear button. Unfortunately there's an unexpected snag. When we use the command option the function specified

must take no arguments. When we use the bind function to do the same job the bound function must take one argument. Thus we need to create a new function with a single parameter which calls `evClear`. Add the following after the `evClear` definition:

```
def evHotKey(event):
    evClear()
```

And add the following line following the definition of the `eHello` Entry widget:

```
eHello.bind("<Control-c>",evHotKey) # the key definition is case sensitive
```

Run the program again and you can now clear the text by either hitting the button or typing Ctrl-c. We could also use bind to capture things like mouse clicks or capturing or losing *focus* (that is, making the window active or inactive) or even the window becoming visible (or hidden). See the Tkinter documentation for more information on this. The hardest part is usually figuring out the format of the event description!

A Short Message

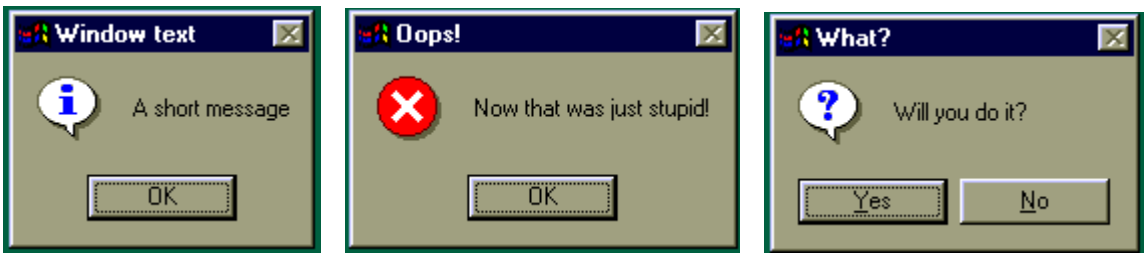
You can report short messages to your users using a *MessageBox*. This is very easy in Tk and is accomplished using the `tkMessageBox` module functions as shown:

```
import tkMessageBox
tkMessageBox.showinfo("Window Text", "A short message")
```

There are also error, warning, Yes/No and OK/Cancel boxes available via different `showXXX` functions. They are distinguished by different icons and buttons. The latter two use `askXXX` instead of `showXXX` and return a value to indicate which button the user pressed, like so:

```
res = tkMessageBox.askokcancel("Which?", "Ready to stop?")
print res
```

Here are some of the Tkinter message boxes:



There are also standard dialog boxes that you can use to get filenames or directory names from the user that look just like the normal GUI "Open File" or "Save File" dialogs. I won't describe them here but you will find examples in the Tkinter reference pages under Standard Dialogs. You will notice there that there are direct equivalents to the EasyGui dialogs available within Tkinter, so you only really need to use EasyGUI when working in a command line environment and want to give a bit of GUI like polish to your program.

Wrapping Applications as Objects

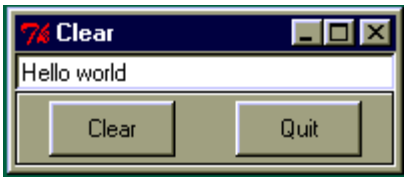
It's common when programming GUI's to wrap the entire application as a class. This begs the question, how do we fit the widgets of a Tkinter application into this class structure? There are two choices, we either decide to make the application itself as a subclass of a Tkinter Frame or have a member field store a reference to the top level window. The latter approach is the one most commonly used in other toolkits so that's the approach we'll use here. If you want to see the first approach in action go back and look at the example in the Event Driven Programming topic. (That example also illustrates the basic use of the incredibly versatile Tkinter Text widget as well as another example of using `bind`)

I will convert the example above, using an Entry field, a Clear button and a Quit button, to an OO structure. First we create an Application class and within the constructor assemble the visual parts of the GUI.

We assign the resultant Frame to `self.mainWindow`, thus allowing other methods of the class access to the top level Frame. Other widgets that we may need to access (such as the Entry field) are likewise assigned to member variables of the application. Using this technique the event handlers become methods of the application class and all have access to any other data members of the application (although in this case there are none) through the `self` reference. This provides seamless integration of the GUI with the underlying application objects:

```
from tkinter import *

# create the event handler to clear the text
class ClearApp:
    def __init__(self, parent=0):
        # create the top level window/frame
        self.mainWindow = Frame(parent)
        self.eHello = Entry(self.mainWindow)
        self.eHello.insert(0,"Hello world")
        self.eHello.pack(fill="x", padx=5, pady=5)
        self.eHello.bind("", self.evHotKey)

        # Now the frame with the buttons.
        fButtons = Frame(self.mainWindow, height=2)
        self.bClear = Button(fButtons, text="Clear",
                             width=10, height=1,command=self.evClear)
        self.bQuit  = Button(fButtons, text="Quit",
                             width=10, height=1, command=self.mainWindow.quit)
        self.bClear.pack(side="left", padx=15, pady=1)
        self.bQuit.pack(side="right", padx=15, pady=1)
        fButtons.pack(side="top", pady=2, fill="x")
        self.mainWindow.pack()
        self.mainWindow.master.title("Clear Application")

    def evClear(self):
        self.eHello.delete(0,END)

    def evHotKey(self, event):
        self.evClear()

# Now create the app and run the eventloop
top = Tk()
app = ClearApp(top)
top.mainloop()
```

Here's the result:



The result looks remarkably like the previous incarnation although I have tweaked some of the configuration and pack options to look more similar to the wxPython example below.

Of course it's not just the main application that we can wrap up as an object. We could create a class based around a Frame containing a standard set of buttons and reuse that class in building dialog windows say. We could even create whole dialogs and use them across several projects. Or we can extend the capabilities of the standard widgets by subclassing them - maybe to create a button that changes colour depending on its state. This is what has been done with the Tix module which is an extension to Tkinter which is also part of the stand library.

Since version 3.1 Tkinter also includes some new features, known as *themed* widgets which greatly improve the look of Tkinter so that it is virtually indistinguishable from the native OS widgets. I won't cover these here but you can read about them on the Tcl/Tk web site.

An alternative - wxPython

There are many other GUI toolkits available but one of the most popular is the wxPython toolkit which is, in turn, a wrapper for the C++ toolkit wxWidgets. wxPython is much more typical than Tkinter of GUI toolkits in general. It also provides more standard functionality than Tk "out of the box" - things like tooltips, status bars etc which have to be hand crafted in Tkinter. We'll use wxPython to recreate the simple "Hello World" Label and Button example above.

One major snag is that wxPython is not yet available for Python v3! It is intended to port it but the maintainer has not, at the time of writing, done so. That means you will need to treat the v2 code below as a reading exercise only.

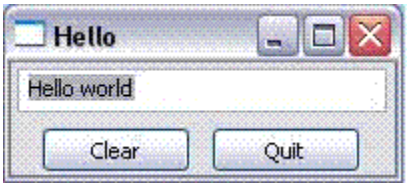
I won't go through this in detail, if you do want to know more about how wxPython works you will need to download the package from the wxPython website.

In general terms the toolkit defines a framework which allows us to create windows and populate them with controls and to bind methods to those controls. It is fully object oriented so you should use methods rather than functions. The example looks like this:

```
import wx

# --- Define a custom Frame, this will become the main window ---
class HelloFrame(wx.Frame):
   def __init__(self, parent, id, title, pos, size):
        wx.Frame.__init__(self, parent, id, title, pos, size)
       # we need a panel to get the right background
        panel = wx.Panel(self)

        # Now create the text and button widgets
        self.tHello = wx.TextCtrl(panel, -1, "Hello world", pos=(3,3), size=(185,
        bClear = wx.Button(panel, -1, "Clear", pos=(15, 32))
        self.Bind(wx.EVT_BUTTON, self.OnClear, bClear)
        bQuit = wx.Button(panel, -1, "Quit", pos=(100, 32))
        self.Bind(wx.EVT_BUTTON, self.OnQuit, bQuit)

   # these are our event handlers
```

```
    def OnClear(self, event):
        self.tHello.Clear()

    def OnQuit(self, event):
        self.Destroy()


# --- Define the Application Object ---
# Note that all wxPython programs MUST define an
# application class derived from wx.App
class HelloApp(wx.App):
    def OnInit(self):
        frame = HelloFrame(None, -1, "Hello", (200,50), (200,90) )
        frame.Show(True)
        self.SetTopWindow(frame)
        return True

# create instance and start the event loop
HelloApp().MainLoop()
```

And it looks like this:



Points to note are the use of a naming convention for the methods that get called by the framework - `OnXXXX`. Also note the `EVT_XXX` constants used to bind events to widgets - there is a whole family of these. wxPython has a vast array of widgets, far more than Tkinter, and with them you can build quite sophisticated GUIs. Unfortunately they tend to use a coordinate based placement scheme which becomes very tedious after a while. It is possible to use a scheme very similar to the Tkinter packer but its not so well documented.

Incidentally it might be of interest to note that this and the very similar Tkinter example above have both got about the same number of lines of executable code - Tkinter: 23, wxPython: 21.

In conclusion, if you just want a quick GUI front end to a text based tool then Tkinter should meet your needs with minimal effort. If you want to build full featured cross platform GUI applications look more closely at wxPython.

Other toolkits include MFC and .NET and of course there is the venerable curses which is a kind of text based GUI! Many of the lessons we've learned with Tkinter apply to all of these toolkits but each has its own characteristics and foibles. Pick one, get to know it and enjoy the wacky world of GUI design. Finally I should mention that many of the toolkits do have graphical GUI builder tools, for example Qt has Blackadder and GTK has Glade. wxPython has Python Card which tries to simplify the whole wxPython GUI building process. There is also a free GUI builder, Boa Constructor, available although still only in Alpha release state. There is even a GUI builder for Tkinter called SpecTix , based on an earlier Tcl tool for building Tk interfaces, but capable of generating code in multiple languages including Python. There is also an enhanced set of widgets for Tkinter called Tix which has recently been added to the standard Python library (and another popular add-in is the Python Mega-Widgets (PMW)) to fill the gap between the basic Tkinter set and those provided by wxPython etc.

That's enough for now. This wasn't meant to be a Tkinter reference page, just enough to get you started. See the Tkinter section of the Python web pages for links to other Tkinter resources.

There are also several books on using Tcl/Tk and several Python books have chapters on Tkinter. I will however come back to Tkinter in the case study, where I illustrate one way of encapsulating a batch mode program in a GUI for improved usability.

Things to remember

- GUIs controls are known as widgets
- Widgets are assembled in a containment hierarchy
- Different GUI toolkits provide different sets of widgets, although there will be a basic set you can assume will be present
- Frames allow you to group related widgets and form the basis of reusable GUI components
- Event handling functions or methods are associated with widgets by linking their name with the widgets `command` property.
- OOP can simplify GUI programming significantly by creating objects that correspond to widget groups and methods that correspond to events.

Previous Next Contents

If you have any questions or feedback on this page send me mail at: alan.gauld@btinternet.com

Recursion

What will we cover?

- A definition of recursion
- How recursion works
- How recursion helps simplify some hard problems

Note: This is a fairly advanced topic and for most applications you don't need to know anything about it. Occasionally, it is so useful that it is invaluable, so I present it here for your study. Just don't panic if it doesn't make sense straight away.

What is it?

Despite what I said earlier about looping being one of the cornerstones of programming it is in fact possible to create programs without an explicit loop construct. Some languages, such as Scheme, do not in fact have an explicit loop construct like `For, While,` etc. Instead they use a technique known as *recursion* . This turns out to be a very powerful technique for some types of problem, so we'll take a look at it now.

Recursion simply means applying a function as a part of the definition of that same function. Thus the definition of GNU (the source of much free software) is said to be recursive because GNU stands for 'GNU's Not Unix'. ie GNU is part of the definition of GNU!

The key to making this work is that there **must be a terminating condition** such that the function branches to a non-recursive solution at some point. (The GNU definition fails this test and so gets stuck in an infinite loop).

Let's look at a simple example. The mathematical factorial function is defined as being the product of all the numbers up to and including the argument, and the factorial of 1 is 1. Thinking about this, we see that another way to express this is that the factorial of N is equal to N times the factorial of (N-1).

Thus:

```
1! = 1
2! = 1 x 2 = 2
3! = 1 x 2 x 3 = 2! x 3 = 6
N! = 1 x 2 x 3 x .... (N-2) x (N-1) x N = (N-1)! x N
```

So we can express this in Python like this:

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

Now because we decrement N each time and we test for N equal to 1 the function must complete. There is a small bug in this definition however, if you try to call it with a number less than 1 it goes into an infinite loop! To fix that change the test to use "<=" instead of "==". This goes to show how easy it is to make mistakes with terminating conditions, this is probably the single most common cause of bugs in recursive functions. Make sure you test all the values around your terminating point to ensure correct operation.

Let's see how that works when we execute it. Notice that the return statement returns `n * (the result of the next factorial call)` so we get:

```
factorial(4) = 4 * factorial(3)
factorial(3) = 3 * factorial(2)
factorial(2) = 2 * factorial(1)
factorial(1) = 1
```

So Python now works its way back up substituting the values:

```
factorial(2) = 2 * 1 = 2
factorial(3) = 3 * 2 = 6
factorial(4) = 4 * 6 = 24
```

Writing the factorial function without recursion actually isn't that difficult, try it as an exercise. Basically you need to loop over all the numbers up to N multiplying as you go. However as we'll see below some functions are much harder to write without recursion.

Recursing over lists

The other area where recursion is very useful is in processing lists. Provided we can test for an empty list, and generate a list minus its first element we can use recursion easily. In Python we do that using a technique called *slicing*. This is explained fully in the Python manual but for our purposes all you need to know is that using an "index" of `[1:]` on a list returns all of the elements from 1 to the end of the list. So to get the first element of a list called L:

```
first = L[0] # just use normal indexing
```

And to get the rest of the list:

```
butfirst = L[1:] # use slicing to get elements 1,2,3,4...
```

Let's try it out at the Python prompt, just to reassure ourselves that it works:

```
>>> L =[1,2,3,4,5]
>>> print( L[0] )
1
>>> print( L[1:] )
[2,3,4,5]
```

OK, let's get back to using recursion to print lists. Consider the trivial case of printing each element of a list of strings using a function printList:

```
def printList(L):
    if L:
        print( L[0] )
        printList(L[1:])
```

If L is true - non empty - we print the first element then process the rest of the list like this:

```
# NON PYTHON PSEUDO CODE
PrintList([1,2,3])
```

```
   prints [1,2,3][0] => 1
   runs printList([1,2,3][1:]) => printList([2,3])
   => we're now in printList([2,3])
         prints [2,3][0] => 2
         runs printList([2,3][1:]) => printList([3])
         => we are now in printList([3])
               prints [3][0] => 3
               runs printList([3][1:]) => printList([])
               => we are now in printList([])
                     "if L" is false for an empty list, so we return None
         => we are back in printList([3])
               it reaches the end of the function and returns None
   => we are back in printList([2,3])
         it reaches the end of the function and returns None
=> we are back in printList([1,2,3])
   it reaches the end of the function and returns None
```

[Note: The above explanation is adapted from one given by Zak Arntson on the Python Tutor mailing list, July 2003]

For a simple list that's a trivial thing to do using a simple for loop. But consider what happens if the List is complex and contains other lists within it. If we can test whether an item is a List using the built-in `type()` function and if it is a list then we can call `printList()` recursively. If it wasn't a list we simply print it. Let's try that:

```
def printList(L):
    # if its empty do nothing
    if not L: return
    # if it's a list call printList on 1st element
    if type(L[0]) == type([]):
        printList(L[0])
    else: #no list so just print
        print( L[0] ) # now process the rest of L
    printList( L[1:] )
```

Now if you try to do that using a conventional loop construct you'll find it very difficult. Recursion makes a very complex task comparatively simple.

There is a catch (of course!). Recursion on large data structures tends to eat up memory so if you are short of memory, or have very large data structures to process the more complex conventional code may be safer.

Finally, both VBScript and JavaScript support recursion too. However since there is little to say that has not already been said I will leave you with a recursive version of the factorial function in each language:

```
<script type="text/vbscript">
Function factorial(N)
   if N <=1 Then
      Factorial = 1
   Else
      Factorial = N * Factorial(N-1)
   End If
End Function

Document.Write "7! = " & CStr(Factorial(7))
</script>
```

```
<script type="text/javascript">
function factorial(n){
   if (n <= 1)
      return 1;
   else
      return n * factorial(n-1);
};

document.write("6! = " + factorial(6));
</script>
```

OK, let's now take another leap into the unknown as we introduce Functional Programming.

Things to Remember

- Recursive functions call themselves within their own definition
- Recursive functions must have a non-recursive terminating condition or an infinite loop will occur.
- Recursion is often, but not always, memory hungry

Previous Next Contents

If you have any questions or feedback on this page send me mail at: alan.gauld@btinternet.com

Functional Programming

What will we Cover?

- The difference between Functional and more traditional programming styles
- Python FP functions and techniques
- Lambda functions
- Short Circuit Boolean evaluation and conditional expressions
- Programs as expressions

In this topic we look at how Python can support yet another programming style: Functional Programming(FP). As with Recursion this is a genuinely advanced topic which you may wish to ignore for the present. Functional techniques do have some uses in day to day programming and the supporters of FP believe it to be a fundamentally better way to develop software.

What is Functional Programming?

Functional programming should not be confused with imperative (or procedural) programming. Neither is it like object oriented programming. It is something different. Not radically so, since the concepts that we will be exploring are familiar programming concepts, just expressed in a different way. The philosophy behind how these concepts are applied to solving problems is also a little different.

Functional programming is all about *expressions*. In fact another way to describe FP might be to term it *expression oriented programming* since in FP everything reduces to an expression. You should recall that an expression is a collection of operations and variables that results in a single value. Thus `x == 5` is a boolean expression. `5 + (7-Y)` is an arithmetic expression. And `"Hello world".uppercase()` is a string expression. The latter is also a function call (Or more strictly a method call) on the string object `"Hello world"` and, as we shall see, functions are very important in FP (You might already have guessed that from the name!).

Functions are used as objects in FP. That is they are often passed around within a program in much the same way as other variables. We have seen examples of this in our GUI programs where we assigned the name of a function to the `command` attribute of a Button control. We treated the event handler function as an object and assigned a reference to the function to the Button. This idea of passing functions around our program is key to FP.

Functional Programs also tend to be heavily List oriented.

Finally FP tries to focus on the *what* rather than the *how* of problem solving. That is, a functional program should describe the problem to be solved rather than focus on the mechanism of solution. There are several programming languages which aim to work in this way, one of the most widely used is Haskell and the Haskell web site (www.haskell.org) has numerous papers describing the philosophy of FP as well as the Haskell language. (My personal opinion is that this goal, however laudable, is rather overstated by FP's advocates.)

A pure functional program is structured by defining an expression which captures the intent of the program. Each term of the expression is in turn a statement of a characteristic of the problem (maybe encapsulated as another expression) and the evaluation of each of these terms eventually yields a solution.

Well, that's the theory. Does it work? Yes, sometimes it works very well. For some types of problem it is a natural and powerful technique. Unfortunately for many other problems it requires a fairly abstract thinking style, heavily influenced by mathematical principles. The resultant code is often far from readable to the layman programmer. The resultant code is also very often much shorter than the equivalent imperative code and more reliable.

It is these latter qualities of conciseness and reliability that have drawn many conventional imperative or object oriented programmers to investigate FP. Even if not embraced whole heartedly there are several powerful tools that can be used by all.

FP and Reliability

The reliability of Functional Programs comes in part from the very close relationship between FP constructs and formal specification languages such as Z or VDM. If a problem is specified in a formal language it is a fairly straightforward step to translate the specification into an FP language like Haskell. Of course if the original specification is wrong then the resultant program will merely accurately reflect the error!

This principle is known in computer science as *"Garbage In, Garbage Out"*. The inherent difficulty of expressing system requirements in a concise and unambiguous manner remains one of the greatest challenges of software engineering.

How does Python do it?

Python provides several functions which enable a functional approach to programming. These functions are all convenience features in that they can be written in Python fairly easily. What is more important however is the *intent* implicit in their provision, namely to allow the Python programmer to work in a FP manner if he/she wishes.

We will look at some of the functions provided and see how they operate on some sample data structures that we define as:

```
spam = ['pork','ham','spices']
numbers = [1,2,3,4,5]

def eggs(item):
    return item
```

map(aFunction, aSequence)

This function applies a Python function, `aFunction` to each member of `aSequence`. The expression:

```
L = map(eggs, spam)
print( list(L) )
```

Results in a new list (in this case identical to spam) being returned in L. Notice that we passed the function `eggs()` into the `map()` function as a value. (That is we didn't use parentheses to execute the function code, we just used its name as a reference to the function.) You might recall we did the same thing with event handlers in the GUI Topic. This ability to treat functions as values is one of the key features of Functional Programming.

We could have achieved the same effect by writing:

```
for i in spam:
   L.append(i)
print( L )
```

Notice however, that the `map` function allows us to remove the need for a nested block of code. From one point of view that reduces the complexity of the program by one level. We'll see that as a recurring theme of FP; that use of the FP functions reduces the relative complexity of the code by eliminating blocks.

filter(aFunction, aSequence)

As the name suggests `filter` extracts each element in the sequence for which the function returns `True`. Consider our list of numbers. If we want to create a new list of only odd numbers we can produce it like so:

```
def isOdd(n): return (n%2 != 0) # use mod operator
L = filter(isOdd, numbers)
print( list(L) )
```

Again notice that we pass the name of the `isOdd` function into `filter` as an argument value rather than calling `isOdd()` as a function.

Alternatively we can write:

```
def isOdd(n): return (n%2 != 0)
for i in numbers:
   if isOdd(i):
      L.append(i)
print( L )
```

Again notice that the conventional code requires two levels of indentation to achieve the same result. Again the increased indentation is an indication of increased code complexity.

There are a few other functional programming tools in a module called `functools` which you might like to import and explore at the >>> prompt. (Remember `dir()` and `help()` are your friends.)

lambda

One feature you may have noticed in the examples so far is that the functions passed to the FP functions tend to be very short, often only a single line of code. To save the effort of defining lots of very small functions Python provides another aid to FP - `lambda`. The name lambda comes from a branch of mathematics called *Lambda Calculus* which uses the Greek letter Lambda to represent a similar concept.

Lambda is a term used to refer to an *anonymous function*, that is, a block of code which can be executed as if it were a function but without a name. Lambdas can be defined anywhere within a program that a legal Python expression can occur, which means we can use them inside our FP functions.

A Lambda looks like this:

```
lambda <aParameterList> : <a Python expression using the parameters>
```

Thus the `isOdd` function above could be rewritten as:

```
isOdd = lambda j: j%2 != 0
```

And we can avoid the definition line completely by creating the `lambda` within the call to `filter`, like so:

```
L = filter(lambda j: j%2 != 0, spam)
print( list(L) )
```

And the call to map can be done using:

```
L = map(lambda n: n, numbers)
print( list(L) )
```

By the way, you may have noticed that we have been explicitly converting the results of `map()` and `filter()` to lists. This is because `map` and `filter` are actually classes that return instances of something called an *iterator*. An iterator is basically something that acts like a list when used in a loop and can be converted to a list but is more efficient in the way it uses memory. Our old friend `range()` is also an iterator. Python makes it possible to create your own iterators, should you need to, but I won't be discussing that in the tutorial.

List Comprehension

List comprehension is a technique for building new lists borrowed from Haskell and introduced in Python since version 2.0. It has a slightly obscure syntax, similar to mathematical set notation. It looks like this:

```
[<expression> for <value> in <collection> if <condition>]
```

Which is equivalent to:

```
L = []
for value in collection:
    if condition:
        L.append(expression)
```

As with the other FP constructs this saves some lines and two levels of indentation. Let's look at some practical examples.

First let's create a list of all the even numbers:

```
>>> [n for n in range(10) if n % 2 == 0 ]
[0, 2, 4, 6, 8]
```

That says we want a list of values (`n`) where `n` is selected from the range 0-9 and `n` is even(`n % 2 == 0`).

The condition at the end could, of course, be replaced by a function, provided the function returns a value that Python can interpret as boolean. Thus looking again at the previous example we could rewrite it as:

```
>>>def isEven(n): return ((n%2) == 0)
>>> [ n for n in range(10) if isEven(n) ]
```

```
[0, 2, 4, 6, 8]
```

Now let's create a list of the squares of the first 5 numbers:

```
>>> [n*n for n in range(5)]
[0, 1, 4, 9, 16]
```

Notice that the final if clause is not needed in every case. Here the initial expression is `n*n` and we use all of the values from the range.

Finally let's use an existing collection instead of the range function:

```
>>> values = [1, 13, 25, 7]
>>> [x for x in values if x < 10]
[1, 7]
```

This could be used to replace the following filter function:

```
>>> print( list(filter(lambda x: x < 10, values)) )
[1, 7]
```

List comprehensions are not limited to one variable or one test however the code starts to become very complex as more variables and tests are introduced.

Whether comprehensions or the traditional functions seem most natural or appropriate to you is purely subjective. When building a new collection based on an existing collection you can use either the previous FP functions or the new list comprehensions. When creating a completely new collection it is usually easier to use a comprehension.

Remember though that while these constructs may seem appealing, the expressions needed to get the desired result can become so complex that it's easier to just expand them out to their traditional python equivalents. There is no shame in doing so - readability is always better than obscurity, especially if the obscurity is just for the sake of being clever!

Other constructs

Of course while these functions are useful in their own right they are not sufficient to allow a full FP style within Python. The control structures of the language also need to be altered, or at least substituted, by an FP approach. One way to achieve this is by applying a side effect of how Python evaluates boolean expressions.

Short Circuit evaluation

Because Python uses *short circuit evaluation* of boolean expressions certain properties of these expressions can be exploited. To recap on short-circuit evaluation: when a boolean expression is evaluated the evaluation starts at the left hand expression and proceeds to the right, stopping when it is no longer necessary to evaluate any further to determine the final outcome.

Taking some specific examples let's see how short circuit evaluation works:

```
>>> def TRUE():
...   print( 'TRUE' )
```

```
...   return True
...
>>>def FALSE():
...   print( 'FALSE' )
...   return False
...
```

First we define two functions that tell us when they are being executed and return the value of their names. Now we use these to explore how boolean expressions are evaluated (Note that the upper case output is from the functions whereas the mixed case output is the result of the expression):

```
>>>print( TRUE() and FALSE() )
TRUE
FALSE
False
>>>print( TRUE() and TRUE() )
TRUE
TRUE
True
>>>print( FALSE() and TRUE() )
FALSE
False
>>>print( TRUE() or FALSE() )
TRUE
True
>>>print( FALSE() or TRUE() )
FALSE
TRUE
True
>>>print( FALSE() or FALSE() )
FALSE
FALSE
False
```

Notice that only **IF** the first part of an *AND* expression is `True` then and only then will the second part be evaluated. If the first part is `False` then the second part will not be evaluated since the expression as a whole **cannot** be `True`.

Likewise in an **OR** based expression if the first part is `True` then the second part need not be evaluated since the whole **must** be `True`.

There is one other feature of Python's evaluation of boolean expressions that we can take advantage of, namely that when evaluating an expression Python does not simply return `True` or `False`, rather it returns the actual value of the expression. Thus if testing for an empty string (which would count as `False`) like this:

```
if "This string is not empty": print( "Not Empty" )
else: print( "No string there" )
```

Python just returns the string itself!

We can use these properties to reproduce branching like behavior. For example suppose we have a piece of code like the following:

```
if TRUE(): print( "It is True" )
else: print( "It is False" )
```

We can replace that with the FP style construct:

```
V =  (TRUE() and "It is True") or ("It is False")
print( V )
```

Try working through that example and then substitute the call to `TRUE()` with a call to `FALSE()`.

Thus by using short circuit evaluation of boolean expressions we have found a way to eliminate conventional if/else statements from our programs. However, these tricks can backfire so in recent versions of Python a new construct has been introduced that allows us to write an if/else condition as an expression, and it is called a *conditional expression*, and it looks like this:

```
result = <True expression> if <test condition> else <False expression>
```

And a real example looks like this:

```
>>> print( "This is True" if TRUE() else "This is not printed" )
TRUE
This is True
```

And using the else:

```
>>> print( "This is True" if FALSE() else "We see it this time" )
FALSE
We see it this time
```

You may recall that in the recursion topic we observed that recursion could be used to replace the loop construct. Thus combining recursion with conditional expressions can remove all conventional control structures from our program, replacing them with pure expressions. This is a big step towards enabling pure FP style solutions.

To put all of this into practice let's write a completely functional style factorial program using `lambda` instead of `def`, recursion instead of a loop and a conditional expression instead of the usual `if/else:`

```
>>> factorial = lambda n: 1 if (n <= 1) else (factorial(n-1) * n)
>>> print( factorial(5) )
120
```

And that really is all there is to it. It may not be quite so readable as the more conventional Python code but it does work and is a purely functional style function in that it is a pure expression.

Conclusions

At this point you may be wondering what exactly is the point of all of this? You would not be alone. Although FP appeals to many Computer Science academics (and often to mathematicians) most practicing programmers seem to use FP techniques sparingly and in a kind of hybrid fashion mixing it with more traditional imperative styles as they feel appropriate.

When you have to apply operations to elements in a list such that `map` or `filter` seem the natural way to express the solution then by all means use them. Just occasionally you may even find that recursion is more appropriate than a conventional loop. Even more rarely will you find a use for short

circuit evaluation or, better, a conditional expression rather than conventions if/else - particularly if required within an expression. As with any programming tool, don't get carried away with the philosophy, rather use whichever tool is most appropriate to the task in hand. At least you know that alternatives exist!

There is one final point to make about `lambda`. There is one area outside the scope of FP that lambda finds a real use and that's for defining event handlers in GUI programming. Event handlers are often very short functions, or maybe they simply call some larger function with a few hard-wired argument values. In either case a lambda function can be used as the event handler which avoids the need to define lots of small individual functions and fill up the name space with names that would only be used once. Remember that a lambda statement returns a function object. This function object is the one passed to the widget and is called at the time the event occurs. If you recall how we define a Button widget in Tkinter, then a lambda would appear like this:

```
def write(s): print( s )
b = Button(parent, text="Press Me",
           command = lambda : write("I got pressed!"))
b.pack()
```

Of course in this case we could have done the same thing by just assigning a default parameter value to `write()` and assigning `write` to the `command` value of the `Button`. However even here using the `lambda` form gives us the advantage that the single `write()` function can now be used for multiple buttons just by passing a different string from the `lambda`. Thus we can add a second button:

```
b2 = Button(parent, text="Or Me",
           command = lambda : write("So did I!"))
b2.pack()
```

We can also employ `lambda` when using the bind technique, which sends an event object as an argument:

```
b3 = Button(parent, text="Press me as well")
b3.bind(<Button-1>, lambda ev : write("Pressed"))
```

Well, that really is that for Functional Programming. There are lots of other resources if you want to look deeper into it, some are listed below. Neither VBScript nor JavaScript directly support FP but both can be used in a functional style by a determined programmer. (In fact many of the deeper aspects of JavaScript are functional by nature.) The key being to structure your programs as expressions and not to allow side-effects to modify program variables.

Other resources

- There is an excellent article by David Mertz on the IBM web site about FP in Python. It goes into more detail about control structures and provides more detailed examples of the concept.
- Other languages support FP even better than Python. Examples include:: Lisp, Scheme, Haskell, ML and some others. The Haskell web site in particular includes a wealth of information about FP.
- There is also a newsgroup, `comp.lang.functional` where you can catch up on the latest happenings and find a useful FAQ.

- There are several book references to be found on the above reference sites. One classic book, which is not entirely about FP but does cover the principles well is *Structure & Interpretation of Computer Programs* by Abelman, Sussman and Sussman. This text focuses on Scheme, a version of Lisp favoured by many academics. My personal primary source has been the book *The Haskell School of Expression* by Paul Hudak which is, naturally enough, about Haskell.

If anyone else finds a good reference drop me an email via the link below.

Things to Remember

- Functional programs are pure expressions
- Python provides `map`, `filter` and `reduce` as well as `list comprehensions` to support FP style programming
- `lambda` expressions are anonymous (i.e. unnamed) blocks of code that can be assigned to variables or used as functions
- Boolean expressions are evaluated only as far as necessary to ensure the result, which fact enables them to be used as control structures
- By combining the FP features of Python with recursion it is possible (but usually not advisable) to write almost any function in an FP style in Python.

Previous Next Contents

If you have any questions or feedback on this page send me mail at: alan.gauld@btinternet.com

A Case Study

For this case study we are going to expand on the word counting program we developed earlier. We are going to create a program which mimics the Unix wc program in that it outputs the number of lines, words and characters in a file. We will go further than that however and also output the number of sentences, clauses and paragraphs. We will follow the development of this program stage by stage gradually increasing its capability then moving it into a module to make it reusable, turning it into an OO implementation for maximum extendability and finally wrapping it in a GUI for ease of use.

Although we will be using Python throughout it would be possible to build JavaScript or VBScript versions of the program with only a little adaptation.

Additional features that could be implemented but will be left as exercises for the reader are to

- calculate the FOG index of the text, where the FOG index can be defined (roughly) as:

  (Average words per sentence) + (Percentage of words more than 5 letters) * (

  and indicates the complexity of the text,
- calculate the number of unique words used and their frequency,
- create a new version which analyzes RTF files.

Counting lines, words and characters

Let's revisit the previous word counter:

```
import string
def numwords(s):
    list = string.split(s)
    return len(list)

inp = open("menu.txt","r")
total = 0

# accumulate totals for each line
for line in inp.readlines():
    total = total + numwords(line)
print "File had %d words" % total

inp.close()
```

We need to add a line and character count. The line count is easy since we loop over each line we just need a variable to increment on each iteration of the loop. The character count is only marginally harder since we can iterate over the list of words adding their lengths in yet another variable.

We also need to make the program more general purpose by reading the name of the file from the command line or if not provided, prompting the user for the name. (An alternative strategy would be to read from standard input, which is what the real wc does.)

So the final wc looks like:

```
import sys, string
```

```
# Get the file name either from the command-line or the user
if len(sys.argv) != 2:
   name = input("Enter the file name: ")
else:
   name = sys.argv[1]

inp = open(name,"r")

# initialize counters to zero; which also creates variables
words, lines, chars = 0, 0, 0

for line in inp:
    lines += 1

    # Break into a list of words and count them
    list = line.split()
    words += len(list)
    chars += len(line) # Use original line which includes spaces etc.

print "%s has %d lines, %d words and %d characters" % (name, lines, words, chars)
inp.close()
```

If you are familiar with the Unix wc command you know that you can pass it a wild-carded filename to get stats for all matching files as well as a grand total. This program only caters for straight filenames. If you want to extend it to cater for wild cards take a look at the glob module and build a list of names then simply iterate over the file list. You'll need temporary counters for each file then cumulative counters for the grand totals. Or you could use a dictionary instead...

Counting sentences instead of lines

When I started to think about how we could extend this to count sentences and words rather than 'character groups' as above, my initial idea was to first loop through the file extracting the lines into a list then loop through each line extracting the words into another list. Finally to process each 'word' to remove extraneous characters.

Thinking about it a little further it becomes evident that if we simply collect the lines we can analyze the punctuation characters to count sentences, clauses etc. (by defining what we consider a sentence/clause in terms of punctuation items). Let's try sketching that in pseudo-code:

```
foreach line in file:
   increment line count
   if line empty:
      increment paragraph count
   count the clause terminators
   count the sentence terminators

report paras, lines, sentences, clauses, groups, words.
```

We will be using regular expressions in the solution here, it may be worth going back and reviewing that topic if you aren't sure how they work. Now lets try turning our pseudo code into real code:

```
import re,sys

# Use Regular expressions to find the tokens
sentenceStops = ".?!"
clauseStops = sentenceStops + ",;:\-" # escape '-' to avoid range effect
sentenceRE = re.compile("[%s]" % sentenceStops)
clauseRE = re.compile("[%s]" % clauseStops)
```

```
# Get file name from commandline or user
if len(sys.argv) != 2:
   name = input("Enter the file name: ")
else:
   name = sys.argv[1]

inp = open(name,"r")
# Now initialize counters
lines, words, chars = 0, 0, 0
sentences,clauses = 0, 0
paras = 1  # assume always at least 1 para

# process file
for line in inp:
   lines += 1
   if line.strip() == "": # empty line
      paras += 1
   words += len(line.split())
   chars += len(line.strip())
   sentences += len(sentenceRE.findall(line))
   clauses += len(clauseRE.findall(line))

# Display results
print '''
The file %s contains:
        %d\t characters
        %d\t words
        %d\t lines in
        %d\t paragraphs with
        %d\t sentences and
        %d\t clauses.
''' % (name, chars, words, lines, paras, sentences, clauses)
```

There are several points to note about this code:

- It uses regular expressions to make the searches most efficient. We could have done the same thing using simple string searches, but we would have needed to search for each punctuation character separately. Regular expressions maximize the efficiency of our program by allowing a single search to find all of the items we want. However regular expressions are also easy to mess up. My first attempt I forgot to escape the '-' character and that then got treated as a range by the regular expression, with the result that any numbers in the file got treated as clause separators! After much head scratching it took a call to the Python community to spot the mistake. A quick '\' character inserted and suddenly all was well again.
- This program is effective in that it does what we want it to do. It is less effective from the re-usability point of view because there are no functions that we can call from other programs, it is not yet a modular program.
- The sentence tests are less than perfect. For example abbreviated titles such as "Mr." will count as a sentence because of the period. We could improve the regular expression by searching for a period, followed by one or more spaces, followed by an uppercase letter, but our "Mr." example will still fail since "Mr." is usually followed by a name which begins with an uppercase letter! This serves to illustrate how difficult it is to *parse* natural languages effectively.

As the case study progresses we will address the second point about re-usability and also start to look at the issues around parsing text in a little more depth, although even by the end we will not have produced a perfect text parser. That is a task that takes us well beyond the sort of programs a beginner might be expected to write.

Turning it into a module

To make the code we have written into a module there are a few basic design principles that we need to follow. First we need to put the bulk of the code into functions so that users of the module can access them. Secondly we need to move the start code (the bit that gets the file name) into a separate piece of code that won't be executed when the function is imported. Finally we will leave the global definitions as module level variables so that users can change their value if they want to.

Let's tackle these items one by one. First move the main processing block into a function, we'll call it `analyze()`. We'll pass a file object into the function as a parameter and the function will return the list of counter values in a tuple.

It will look like this:

```
##############################
# Module: grammar
# Created: A.J. Gauld, 2004,8,8
#
# Function:
# Provides facilities to count words, lines, characters,
# paragraphs, sentences and 'clauses' in text files.
# It assumes that sentences end with [.!?] and paragraphs
# have a blank line between them. A 'clause' is simply
# a segment of sentence separated by punctuation. The
# sentence and clause searches are regular expression
# based and the user can change the regex used. Can also
# be run as a program.
##############################
import re, sys

############################
# initialize global variables
paras = 1 # We will assume at least 1 paragraph!
lines, sentences, clauses, words, chars = 0,0,0,0,0
sentenceMarks = '.?!'
clauseMarks = '&();:,\-' + sentenceMarks
sentenceRE = None  # set via a function call
clauseRE = None
format = '''
The file %s contains:
        %d\t characters
        %d\t words
        %d\t lines in
        %d\t paragraphs with
        %d\t sentences and
        %d\t clauses.
'''


############################
# Now define the functions that do the work

# setCounters allows us to recompile the regex if we change
# the token lists
def setCounterREs():
    global sentenceRE, clauseRE
    sentenceRE = re.compile('[%s] +' % sentenceMarks)
    clauseRE = re.compile('[%s] +' % clauseMarks)
```

```
# reset counters gets called by analyze()
def resetCounters():
     chars, words, lines, sentences, clauses = 0,0,0,0,0
     paras = 1

# reportStats is intended for the driver
# code, it offers a simple text report
def reportStats(theFile):
    print format % (theFile.name, chars, words, lines,
                    paras, sentences, clauses)

# analyze() is the key function which processes the file
def analyze(theFile):
    global chars,words,lines,paras,sentences,clauses
    # check if REs already compiled
    if not (sentenceRE and clauseRE):
        setCounterREs()
    resetCounters()
    for line in theFile:
      lines += 1
      if line.strip() == "": # empty line
         paras += 1
      words += len(line.split())
      chars += len(line.strip())
      sentences += len(sentenceRE.findall(line))
      clauses += len(clauseRE.findall(line))


# Make it run if called from the command line (in which
# case the 'magic' __name__ variable gets set to '__main__'
if __name__ == "__main__":
    if len(sys.argv) != 2:
        print "Usage: python grammar.py <filename>"
        sys.exit()
    else:
        aFile = open(sys.argv[1],"r")
        analyze(aFile)
        reportStats(aFile)
        aFile.close()
```

First thing to notice is the commenting at the top. This is common practice to let readers of the file get an idea of what it contains and how it should be used. The version information (Author and date) is useful too if comparing results with someone else who may be using a more or less recent version.

The final section is a feature of Python that calls any module loaded at the command line `"__main__"`. We can test the special, built-in `__name__` variable and, if it's `main`, we know that the module is not just being imported, but run, and so we execute the trigger code inside the `if` block.

This trigger code includes a user friendly hint about how the program should be run if no filename is provided, or indeed if too many filenames are provided, it could instead - or in addition - ask the user for a filename using `input()`.

Notice that the `analyze()` function uses the initialization functions to make sure the counters and regular expressions are all set up properly before it starts. This caters for the possibility of a user calling analyze several times, possibly after changing the regular expressions used to count clauses and sentences.

Finally note the use of global to ensure that the module level variables get set by the functions, without global we would create local variables and have no effect on the module level ones.

Using the grammar module

Having created a module we can use it as a program at the OS prompt as before by typing:

```
C:\> python grammar.py spam.txt
```

However provided we saved the module in a location where Python can find it, we can also import the module into another program or at the Python prompt. Lets try some experiments based on a test file called spam.txt which we can create and looks like this:

```
This is a file called spam. It has
3 lines, 2 sentences and, hopefully,
5 clauses.
```

Now, let's fire up Python and play a little:

```
>>> import grammar
>>> grammar.setCounterREs()
>>> txtFile = open("spam.txt")
>>> grammar.analyze(txtFile)
>>> grammar.reportStats()
The file spam.txt contains:
        80       characters
        16       words
        3        lines in
        1        paragraphs with
        2        sentences and
        1        clauses.

>>>  # redefine sentences as ending in vowels!
>>> grammar.sentenceMarks = 'aeiou'
>>> grammar.setCounterREs()
>>> grammar.analyze(txtFile)
>>> print grammar.sentences
21
>>> txtFile.close()
```

As you can see redefining the sentence tokens changed the sentence count radically. Of course the definition of a sentence is pretty bizarre but it shows that our module is usable and moderately customizable too. Notice too that we were able to print the sentence count directly, we don't need to use the provided `reportStats()` function. This demonstrates the value of an important design principle, namely *separation of data and presentation*. By keeping the display of data separate from the calculation of the data we make our module much more flexible for our users.

To conclude our course we will rework the grammar module to use OO techniques and then add a simple GUI front end. In the process you will see how an OO approach results in modules which are even more flexible for the user and more extensible too.

Classes and objects

One of the biggest problems for the user of our module is the reliance on global variables. This means that it can only analyze one file at a time, any attempt to handle more than that will result in the global values being over-written.

By moving these globals into a class we can then create multiple instances of the class (one per file) and each instance gets its own set of variables. Further, by making the methods sufficiently granular we can create an architecture whereby it is easy for the creator of a new type of document object to modify the search criteria to cater for the rules of the new type. (eg. by rejecting all HTML tags from the word list we could process HTML files as well as plain ASCII text).

Our first attempt at this creates a Document class to represent the file we are processing:

```
#! /usr/local/bin/python
#################################
# Module: document.py
# Author: A.J. Gauld
# Date:   2004/08/10
# Version: 3.0
#################################
# This module provides a Document class which
# can be subclassed for different categories of
# Document(text, HTML, Latex etc). Text and HTML are
# provided as samples.
#
# Primary services available include
#    - analyze(),
#    - reportStats().
#################################
import sys,re

'''
Provides 2 classes for parsing "text/files.
A Generic Document class for plain ACII text,
and an HTMLDocument for HTML files.
'''

class Document:
  sentenceMarks = '?!.'
  clauseMarks = '&()\-;:,' + sentenceMarks

  def __init__(self, filename):
      self.filename = filename
      self.setREs()
      self.setCounters()

  def setCounters(self):
      self.paras = 1
      self.lines = self.getLines()
      self.sentences, self.clauses, self.words, self.chars = 0,0,0,0

  def setREs(self):
      self.sentenceRE = re.compile('[%s]' % Document.sentenceMarks)
      self.clauseRE = re.compile('[%s]' % Document.clauseMarks)

  def getLines(self):
      infile = open(self.filename)
      lines = infile.readlines()
      infile.close()
      return lines

  def analyze(self):
      self.setCounters()
      for line in self.lines:
          self.sentences += len(self.sentenceRE.findall(line))
          self.clauses += len(self.clauseRE.findall(line))
          self.words += len(line.split())
```

```
          self.chars += len(line.strip())
          if line.strip() == "":
             self.paras += 1

  def formatResults(self):
      format = '''
      The file %s contains:
        %d\t characters
        %d\t words
        %d\t lines in
        %d\t paragraphs with
        %d\t sentences and
        %d\t clauses.
      '''
      return format % (self.filename, self.chars,
                       self.words, len(self.lines),
                       self.paras, self.sentences, self.clauses)

class TextDocument(Document):
  pass

class HTMLDocument(Document):
  pass


if __name__ == "__main__":
       if len(sys.argv) == 2:
         doc = Document(sys.argv[1])
         doc.analyze()
         print doc.formatResults()
       else:
         print "Usage: python document3.py "
         print "Failed to analyze file"
```

There are several points to notice here. First is the use of *class variables* at the beginning of the class definition to store the sentence and clause markers. Class variables are shared by all the instances of the class so they are a good place to store common information. They can be accessed by using the class name, as I've done here, or by using the usual self. I prefer to use the class name because it highlights the fact that they are class variables.

I've also added a new method, `setCounters()` for flexibility when we come to deal with other document types. It's quite likely that we will use a different set of counters when analyzing HTML files - maybe the number of tags for example. By pairing up the `setCounters()` and `formatResults()` methods and providing a new `analyze()` method we can pretty much deal with any kind of document.

The other methods are more stable, reading the lines of a file is pretty standard regardless of file type and setting the two regular expressions is a convenience feature for experimenting, if we don't need to we won't.

As it stands we now have functionality identical to our module version but expressed as a class. But now to really utilize OOP style we need to deconstruct some of our class so that the base level or *abstract*Document only contains the bits that are truly generic. The Text handling bits will move into the more specific, or *concrete* `TextDocument` class. We'll see how to do that next.

Text Document

We are all familiar with plain text documents, but it's worth stopping to consider exactly what we mean by a text document as compared to a more generic concept of a document. Text documents consist of plain ASCII arranged in lines which contain groups of letters arranged as words separated by spaces and other punctuation marks. Groups of lines form paragraphs which are separated by blank lines (other definitions are possible of course, but these are the ones I will use.) A vanilla document is a file comprising lines of ASCII characters but we know very little about the formatting of those characters within the lines. Thus our vanilla document class should really only be able to open a file, read the contents into a list of lines and perhaps return counts of the number of characters and the number of lines. It will provide empty *hook methods* for subclasses of document to implement.

On the basis of what we just described a Document class will look like:

```
##############################
# Module: document
# Created: A.J. Gauld, 2004/8/15
#
# Function:
# Provides abstract Document class to count lines, characters
# and provide hook methods for subclasses to use to process
# more specific document types
##############################

class Document:
   def __init__(self,filename):
       self.filename = filename
       self.lines = self.getLines()
       self.chars = reduce(lambda l1,l2: l1+l2, [len(L) for L in self.lines])
       self._initSeparators()

   def getLines(self):
       f = open(self.filename,'r')
       lines = f.readlines()
       f.close()
       return lines

   # list of hook methods to be overridden
   def formatResults(self):
   return "%s contains $d lines and %d characters" % (len(self.lines),
                                                      self.chars)
   def _initSeparators(self): pass
   def analyze(self): pass
```

Note that the `_initSeparators` method has an underscore in front of its name. This is a style convention often used by Python programmers to indicate a method that should only be called from inside the class's methods, it is not intended to be accessed by users of the object. Such a method is sometimes called *protected* or *private* in other languages.

Also notice that I have used the functional programming function `reduce()` along with a `lambda` function and a `list comprehension` to calculate the number of characters. Recall that `reduce` takes a list and performs an operation (the `lambda`) on the first two members and inserts the result as the first member, it repeats this until only the final result remains which is returned as the final result of the function. In this case the list is the list of lengths of the lines in the file produced by the `comprehension` and so it replaces the first two lengths with their sum and then gradually adds each subsequent length until all the line lengths are processed. Actually, in this case, I could have used the built in function `sum()` to do the same thing, but I wanted to illustrate the functional programming structures being used in a real example.

Finally note that because this is an abstract class we have not provided a runnable option using `if __name__ == etc`

Our text document now looks like:

```
class TextDocument(Document):
   def __init__(self,filename):
       self.paras = 1
       self.words, self.sentences, self.clauses = 0,0,0
       Document.__init__(self, filename)

   # now override hooks
   def formatResults(self):
       format = '''
       The file %s contains:
            %d\t characters
            %d\t words
            %d\t lines in
            %d\t paragraphs with
            %d\t sentences and
            %d\t clauses.
       '''
       return format % (self.filename, self.chars,
                        self.words, len(self.lines),
                        self.paras, self.sentences, self.clauses)

   def _initSeparators(self):
       sentenceMarks = "[.!?]"
       clauseMarks = "[.!?,&:;-]"
       self.sentenceRE = re.compile(sentenceMarks)
       self.clauseRE = re.compile(clauseMarks)

   def analyze(self):
       for line in self.lines:
          self.sentences += len(self.sentenceRE.findall(line))
          self.clauses += len(self.clauseRE.findall(line))
          self.words += len(line.split())
          self.chars += len(line.strip())
          if line.strip() == "":
             self.paras += 1

if __name__ == "__main__":
       if len(sys.argv) == 2:
         doc = TextDocument(sys.argv[1])
         doc.analyze()
         print doc.formatResults()
       else:
         print "Usage: python <document> "
         print "Failed to analyze file"
```

One thing to notice is that this combination of classes achieves exactly the same as our first non-OOP version. Compare the length of this with the original file - building reusable objects is not cheap! Unless you are sure you need to create objects for reuse consider doing a non-OOP version it will probably be less work! However if you do think you will extend the design, as we will be doing in a moment then the extra work will repay itself.

The next thing to consider is the physical location of the code. We could have shown two files being created, one per class. This is a common OOP practice and keeps things well organized, but at the expense of a lot of small files and a lot of import statements in your code when you come to use those classes/files.

An alternative scheme, which I have used, is to treat closely related classes as a group and locate them all in one file, at least enough to create a minimal working program. Thus in our case we have combined our Document and TextDocument classes in a single module. This has the advantage that the working class provides a template for users to read as an example of extending the abstract class. It has the disadvantage that changes to the TextDocument may inadvertently affect the Document class and thus break some other code. There is no clear winner here and even in the Python library there are examples of both styles. Pick a style and stick to it would be my advice.

One very useful source of information on this kind of text file manipulation is the book by David Mertz called *"Text Processing in Python"* and it is available in paper form as well as online, here. Note however that this is a fairly advanced book aimed at professional programmers so you may find it tough going initially, but persevere because there are some very powerful lessons contained within it.

HTML Document

The next step in our application development is to extend the capabilities so that we can analyze HTML documents. We will do that by creating a new class. Since an HTML document is really a text document with lots of HTML tags and a header section at the top we only need to remove those extra elements and then we can treat it as text. Thus we will create a new `HTMLDocument` class derived from `TextDocument`. We will override the `getLines()` method that we inherit from `Document` such that it throws away the header and all the HTML tags.

Thus HTMLDocument looks like:

```
class HTMLDocument(TextDocument):
    def getLines(self):
       lines = TextDocument.getLines(self)
       lines = self._stripHeader(lines)
       lines = self._stripTags(lines)
       return lines

    def _stripHeader(self,lines):
       ''' remove all lines up until start of  element '''
       bodyMark = '<body>'
       bodyRE = re.compile(bodyMark,re.IGNORECASE)
       while bodyRE.findall(lines[0]) == []:
          del lines[0]
       return lines

    def _stripTags(self,lines):
       ''' remove anything between < and >, not perfect but ok for now'''
       tagMark = '<.+>'
       tagRE = re.compile(tagMark)
       lines2 = []
       for line in lines:
           line = tagRE.sub('',line).strip()
           if line: lines2.append(line)
       return lines2
```

Note 1: We have used the inherited method within `getLines`. This is quite common practice when extending an inherited method. Either we do some preliminary processing or, as here, we call the inherited code then do some extra work in the new class. This was also done in the `__init__` method of the `TextDocument` class above.

Note 2: We access the inherited `getLines` method via `TextDocument` not via `Document` (which is where it is actually defined) because (a) we can only 'see' `TextDocument` in our code and (b) `TextDocument` inherits all of `Document`'s features so in effect does have a `getLines` too.

Note 3: The other two methods are notionally private (notice the leading underscore?) and are there to keep the logic separate and also to make extending this class easier in the future, for say an XHTML or even XML document class? You might like to try building one of those as an exercise.

Note 4: It is very difficult to accurately strip HTML tags using regular expressions due to the ability to nest tags and because bad authoring often results in unescaped '<' and '>' characters looking like tags when they are not. In addition tags can run across lines and all sorts of other nasties. A much better way to convert HTML files to text is to use an HTML parser such as the one in the standard `HTMLParser` module. As an exercise rewrite the `HTMLDocument` class to use the parser module to generate the text lines.

To test our HTMLDocument we need to modify the *driver* code at the bottom of the file to look like this:

```
if __name__ == "__main__":
       if len(sys.argv) == 2:
         doc = HTMLDocument(sys.argv[1])
         doc.analyze()
         print doc.formatResults()
       else:
         print "Usage: python <document> "
         print "Failed to analyze file"
```

Adding a GUI

To create a GUI we will use Tkinter which we introduced briefly in the Event Driven Programming section and further in the GUI Programming topic. This time the GUI will be slightly more sophisticated and use a few more of the *widgets* that Tkinter provides.

One thing that will help us create the GUI version is that we took great care to avoid putting any print statements in our classes, the display of output is all done in the driver code. This helps when we come to use a GUI because we can use the same output string and display it in a widget instead of printing it on stdout. The ability to more easily wrap an application in a GUI is a major reason to avoid the use of print statements inside data processing functions or methods.

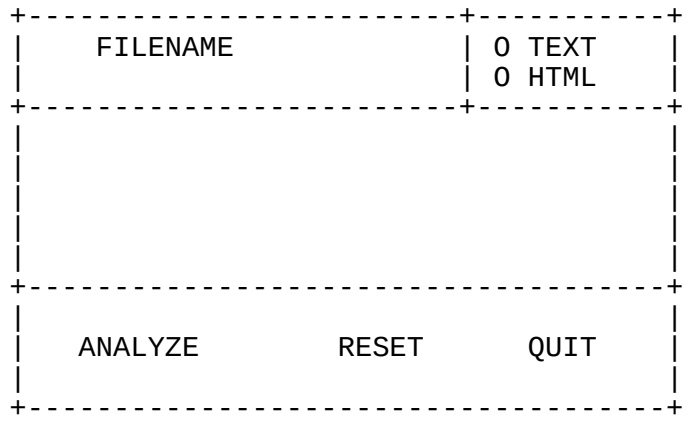
Designing a GUI

The first step in building any GUI application is to try to visualize how it will look. We will need to specify a filename, so it will require an *Edit* or *Entry* control. We also need to specify whether we want textual or HTML analysis, this type of 'one from many' choice is usually represented by a set of *Radiobutton* controls. These controls should be grouped together to show that they are related.

The next requirement is for some kind of display of the results. We could opt for multiple *Label* controls one per counter. Instead I will use a simple text control into which we can insert strings, this is closer to the spirit of the commandline output, but ultimately the choice is a matter of preference by the designer.

Finally we need a means of initiating the analysis and quitting the application. Since we will be using a text control to display results it might be useful to have a means of resetting the display too. These command options can all be represented by *Button* controls.

Sketching these ideas as a GUI gives us something like:



Now we are ready to write some code. Let's take it step by step:

```
from Tkinter import *
import document

################### CLASS DEFINITIONS ######################
class GrammarApp(Frame):
  def __init__(self, parent=0):
    Frame.__init__(self,parent)
    self.type = 2    # create variable with default value
    self.master.title('Grammar counter')
    self.buildUI()
```

Here we have imported the Tkinter and document modules. For the former we have made all of the Tkinter names visible within our current module whereas with the latter we will need to prefix the names with `document.`

We have also defined our application to be a subclass of `Frame` and the `__init__` method calls the `Frame.__init__` superclass method to ensure that Tkinter is set up properly internally. We then create an attribute which will store the document type value and finally call the `buildUI` method which creates all the widgets for us. We'll look at `buildUI()` next:

```
   def buildUI(self):
     # Now the file information: File name and type
     fFile = Frame(self)
     Label(fFile, text="Filename: ").pack(side="left")
     self.eName = Entry(fFile)
     self.eName.insert(INSERT,"test.htm")
     self.eName.pack(side=LEFT, padx=5)


     # to keep the radio buttons lined up with the
     # name we need another frame
     fType = Frame(fFile, borderwidth=1, relief=SUNKEN)
     self.rText = Radiobutton(fType, text="TEXT",
                              variable = self.type, value=2,
                              command=self.doText)
     self.rText.pack(side=TOP, anchor=W)
     self.rHTML = Radiobutton(fType, text="HTML",
                              variable=self.type, value=1,
                              command=self.doHTML)
     self.rHTML.pack(side=TOP, anchor=W)
     # make TEXT the default selection
```

```
        self.rText.select()
        fType.pack(side=RIGHT, padx=3)
        fFile.pack(side=TOP, fill=X)


        # the text box holds the output, pad it to give a border
        # and make the parent the application frame (ie. self)
        self.txtBox = Text(self, width=60, height=10)
        self.txtBox.pack(side=TOP, padx=3, pady=3)


        # finally put some command buttons on to do the real work
        fButts = Frame(self)
        self.bAnal = Button(fButts, text="Analyze",
                            command=self.doAnalyze)
        self.bAnal.pack(side=LEFT, anchor=W, padx=50, pady=2)
        self.bReset = Button(fButts, text="Reset",
                             command=self.doReset)
        self.bReset.pack(side=LEFT, padx=10)
        self.bQuit = Button(fButts, text="Quit",
                            command=self.doQuit)
        self.bQuit.pack(side=RIGHT, anchor=E, padx=50, pady=2)

        fButts.pack(side=BOTTOM, fill=X)
        self.pack()
```

I'm not going to explain all of that. If you've read my GUI topic it should mostly be clear but for more detail I recommend that you take a look at the Tkinter tutorial and reference found on the Pythonware web site. This is an excellent introduction and reference to Tkinter going beyond the basics that I cover in my topic. The general principle is that you create widgets from their corresponding classes, providing options as *named parameters*, then the widget is *packed* into its containing frame.

The other key points to note are the use of subsidiary `Frame` widgets to hold the Radiobuttons and Command buttons. The Radiobuttons also take a pair of options called `variable & value`, the former links the Radiobuttons together by specifying the same external variable (`self.type`) and the latter gives a unique value for each Radiobutton. Also notice the `command=xxx` options passed to the button controls. These are the methods that will be called by Tkinter when the button is pressed. The code for these comes next:

```
    ################## EVENT HANDLING METHODS ####################
    # time to die...
    def doQuit(self):
      self.quit()


    # restore default settings
    def doReset(self):
      self.txtBox.delete(1.0, END)
      self.rText.select()


    # set radio values
    def doText(self):
      self.type = 2

    def doHTML(self):
      self.type = 1
```

These methods are all fairly trivial and hopefully by now are self explanatory. The final event handler is the one which does the analysis:

```
    # Create appropriate document type and analyze it.
    # then display the results in the form
    def doAnalyze(self):
      filename = self.eName.get()
      if filename == "":
         self.txtBox.insert(END,"\nNo filename provided!\n")
         return
      if self.type == 2:
         doc = document.TextDocument(filename)
      else:
         doc = document.HTMLDocument(filename)
      self.txtBox.insert(END, "\nAnalyzing...\n")
      doc.analyze()
      resultStr = doc.formatResults()
      self.txtBox.insert(END, resultStr)
```

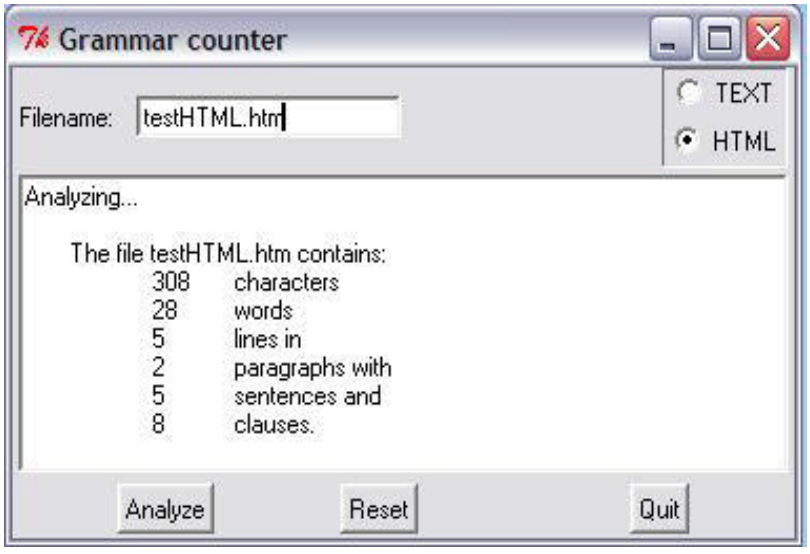
Again you should be able to read this and see what it does. The key points are that:

- it checks for a valid filename before creating the Document object.
- It uses the `self.type` value set by the Radiobuttons to determine which type of Document to create.
- It appends (the `END` argument to `insert`) the results to the Text box which means we can analyze several times and compare results - one advantage of the text box versus the multiple label output approach.

All that's needed now is to create an instance of the `GrammarApp` application class and set the event loop running, we do this here:

```
myApp = GrammarApp()
myApp.mainloop()
```

Let's take a look at the final result as seen under MS Windows, displaying the results of analyzing a test HTML file,



That's it. You can go on to make the HTML processing more sophisticated if you want to. You can create new modules for new document types. You can try swapping the text box for multiple labels packed into a frame. But for our purposes we're done. The next section offers some ideas of where to go next depending on your programming aspirations. The main thing is to enjoy it and always remember: the computer is dumb!

Previous References Contents

If you have any questions or feedback on this page send me mail at: alan.gauld@btinternet.com

Under Construction

HP Web PrintSmart

An error occurred while reading this web site. Please make sure that you have an active Internet connection and that you have entered the URL correctly.