Experiments in Python

Notes:

In the previous sections you have seen how to develop programs using Scratch and Greenfoot. These are a great environments for learning how to construct a program from component parts. They take data from the program and the user, manipulate it and then change what you see on screen by following the instructions in your program.

However, this style of programming has limitations when it comes to constructing solutions to much larger and more complex problems. To get the best performance from a computer and to get the most flexibility, there are many other computer programming languages and techniques. They number in the hundreds, and each has its own place in solving computer-science problems. Your Raspberry Pi could interpret pretty much all of these languages.

To help you move closer to the process by which many real-world applications are developed, this chapter shows you how programs are written in one particularly popular language, called Python. According to Wikipedia, "Python was conceived in the late 1980s and its implementation was started in December 1989 by Guido van Rossum at CWI in the Netherlands." Python is a language that is intended to be fun to use. It is named after the British comedy television series *Monty Python's Flying Circus*. If you don't find it fun, wait until you try some other computer languages!

This chapter is not intended as a tutorial or reference guide to the Python language – if you want these, then there are many fine resources on the internet. However, hopefully, these listings will be a great starting point for you to experiment yourself.

Notes:

From the menu on your desktop, you should be able to find “Programming” and then “Idle3”. Select this to open up a special Python editor for creating Python programs, called “IDLE”. If you cannot find it, press ALT-F2 and then type “Idle3” in the box that appears.

The first thing IDLE gives you is a “**Python Shell**”. Snakes don’t have shells but if Python were a nut, this would be its shell.

Now press the Return key a few times and you’ll see that “>>>” appears at the start of each line. That is called the “**prompt**”. The prompt is Python’s way of telling you it is waiting for you to give it some instructions. Prompts are used in lots of computer-science applications when the computer is waiting for you.

Firstly, you need to type in some commands at the prompt to make your Python shell do something. So type this (don’t type the prompt >>>):

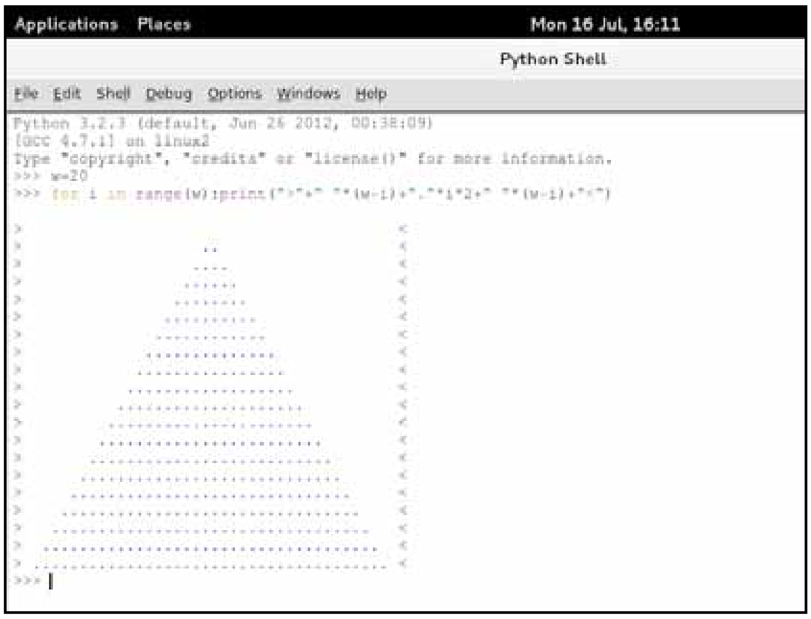
```
>>> w = 20
>>> print (w)
```

This will show you the number 20 in the shell window. “w” is a **variable** that you have set to the value 20. Then you print that value on the output.

Let’s use “w” in a more interesting way. Try this:

```
>>> w = 20
>>> for i in range(w):print(">"+" "*(w-i)+"."*i*2+" "*(w-i)+"<")
```

Use IDLE’s “print” function to create this pyramid.



```
Python 3.2.3 (default, Jun 26 2012, 00:38:09)
[GCC 4.7.1] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> w=20
>>> for i in range(w):print(">"+" "*(w-i)+"."*i*2+" "*(w-i)+"<")

>                                        <
>                   ..                   <
>                  ....                  <
>                 ......                 <
>                ........                <
>               ..........               <
>              ............              <
>             ..............             <
>            ................            <
>           ..................           <
>          ....................          <
>         ......................         <
>        ........................        <
>       ..........................       <
>      ............................      <
>     ..............................     <
>    ................................    <
>   ..................................   <
>  ....................................  <
> ...................................... <
>>> 
```

Press Return twice. Now, that’s a little more impressive. If you want, you can change “w” by typing:

```
>>> w = 35
```

Notes:

Then you can use your arrow keys to move the cursor up to the line beginning with "for" and then press Return, twice. If you are wondering why you need to press Return twice; the first time is to complete the "for" command. When you press the second time, Python recognises you have finished that command and executes what you have typed.

Now let's get some real computer graphics going. Enter this:

```
>>> import pygame
```

After a couple of seconds you'll see the prompt reappear.

```
>>> import pygame
>>>
```

Wow! You imported "PyGame". So what is that, and what does it mean? Again, don't worry about that. You have just told Python that you want to use its game-creation features. "Import" loads extra commands into Python that you can use from now on.

A note about text colours

While you were typing "import pygame" you should have noticed that the word "import" changed colour as soon as you finished typing it. This is because IDLE has a feature called "syntax highlighting". The colouring isn't required for any Python program to work but it is helpful to programmers when reading code. When the text of a program is highlighted with colours, it makes it easier to spot mistakes.

For now, just think of it as the opposite of the little red squiggles that mark spelling errors when you write an essay in a word processor. It doesn't point out errors, but highlights that IDLE has recognised a key Python command.

Making an amazing game will take ages and lots of commands, but while we're building up to that, here are just a few more lines for you to type to see something wonderful. Type these in exactly as you see below.

```
>>> deepblue = (26,0,255)
>>> mintcream = (254,255,250)
```

This has given some names to a couple of "**tuples**" for you to use later. These tuples are just data for the computer, and it will be using them to create colours in just a moment.

```
>>> pygame.init()
>>> size = (500,500)
>>> surface = pygame.display.set_mode(size)
```

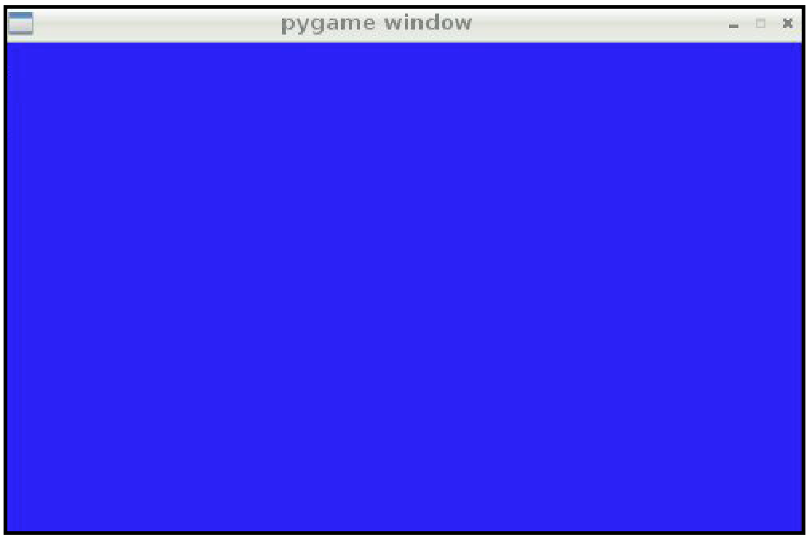
You should now have a display on the screen, which has a surface of 500 x 500 tiny dots, or "**pixels**". That's a grand total of 250,000 pixels. Let's change all 250,000 pixels to a deep blue colour using that "deepblue" tuple we created.

```
>>> surface.fill(deepblue)
```

You have filled the surface with the blue colour, but nothing will happen on screen. The computer does not know you have finished drawing things to the surface, so it doesn't do anything just yet. Let's tell PyGame that it can update the display with this command:

```
>>> pygame.display.update()
```

Update your screen to make the deep blue fill you created appear.



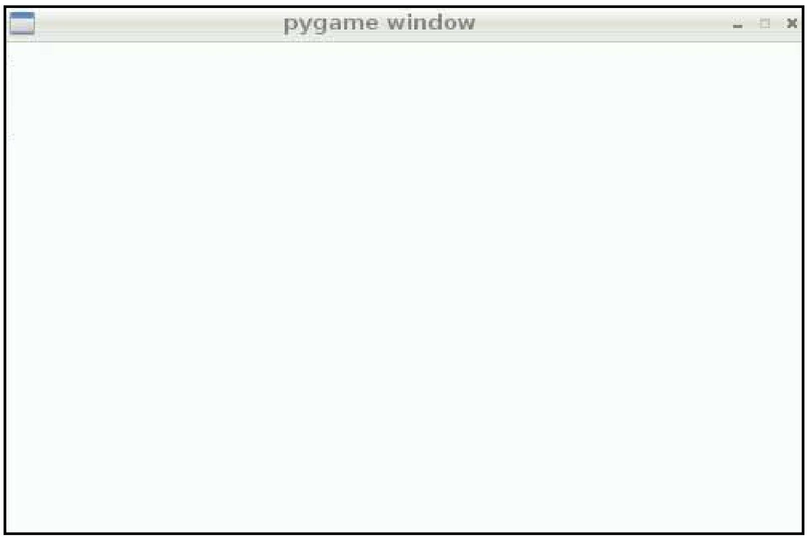
You'll now have a blue surface in your displayed window. So, that was either fun, okay or really dull. Let's hope it was at least okay.

How about using the other colour? Type:

```
>>> surface.fill(mintcream)
>>> pygame.display.update()
```

Hey... that's rather plain.

Update your screen again to make the your mint cream fill appear.



Notes:

Tip...

Pixel is the short form of "picture element". In programming, a pixel can be set to any colour you like, and your "screen" is made up of many rows and columns of them, packed so close together that they blur into one another and look like a single picture.

Notes:

Try making up your own colours.

```
>>> PiColour = (123,456,789)
>>> surface.fill(PiColour)
>>> pygame.display.update()
```

Argh! Now that wasn't fun. You created a bad colour and Python has told you (in American).

```
TypeError: invalid color argument
```

You will have to watch for American spellings like these when writing commands to computers, as a lot of computing standards are developed in the USA.

Python is telling you that the colour tuple is not valid. The PyGame documentation says that the colour tuple must only have values from 0 to 255. Each number represents how much red, green or blue you want in the colour (in that order). Mixing these will give you every colour that the computer can display. That's 256 x 256 x 256 different colours. Try typing this at the prompt to get the answer to that product.

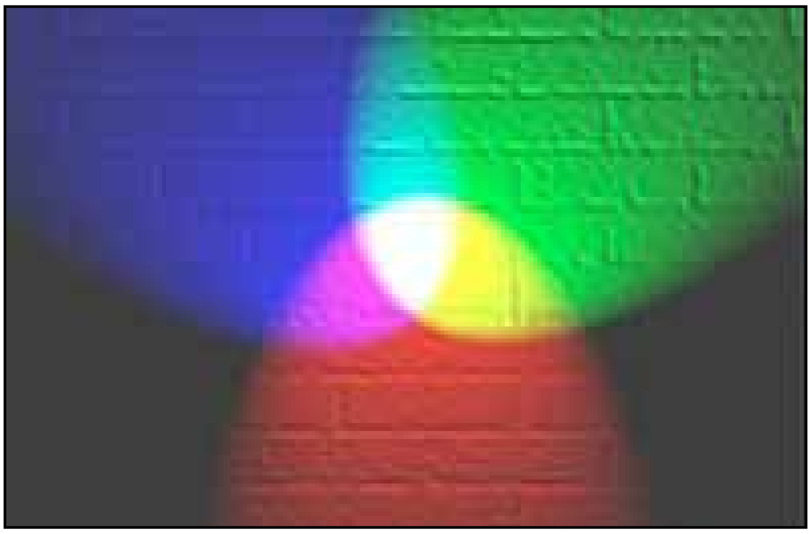
```
>>> 256 * 256 * 256
```

Or even:

```
>>> 256 ** 3
```

The symbols ** mean "raised to the power of". Notice that I haven't used "print()". Python can be a great calculator, too!

The computer can display 256 x 256 x 256 colours. That's 16,777,216 different colours!



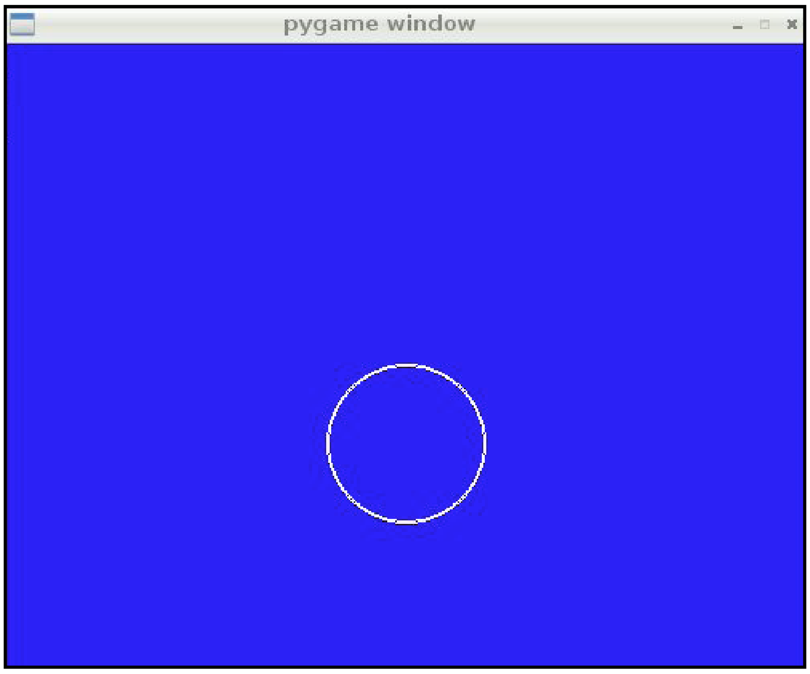
Going round in circles

Now you have filled the background, let's add a circle to the surface.

```
>>> surface.fill(deepblue)
>>> position = (250,250)
>>> radius = 50
>>> linewidth = 2
>>> pygame.draw.circle(surface, mintcream, position, radius,
linewidth)
```

Update again to see the circle added.

```
>>> pygame.display.update()
```



Try changing some of the numbers for drawing the circle and see what happens. Remember to update after each circle you add to the surface.

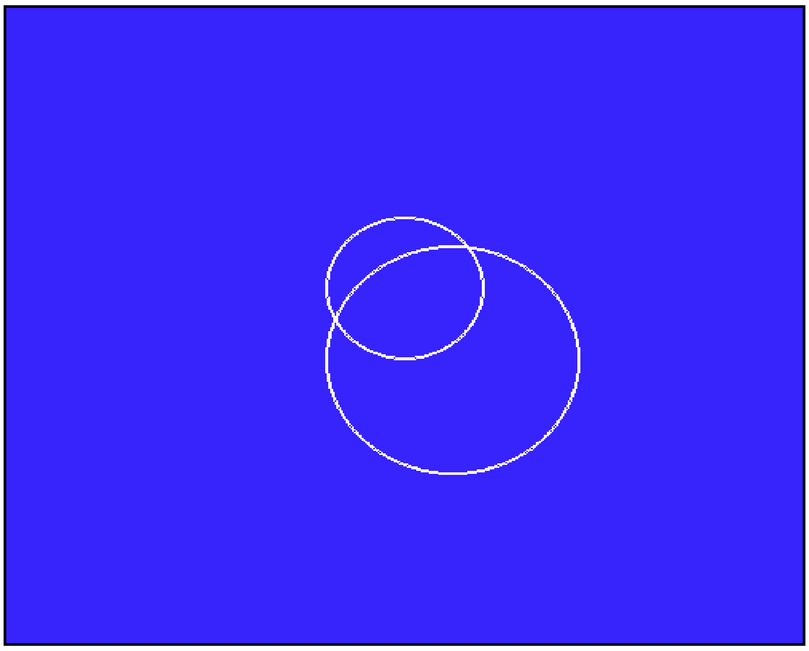
Notes:

Tip...

Sometimes, we have more code to write than fits on one line. When you see a line of code such as the last one here, that goes onto a new line without a prompt or indent, that is really just one line of code. Don't press Return to start a new line, just let it word wrap and Python will work it out for you.

```
>>> position = (280,300)
>>> radius = 80
>>> pygame.draw.circle(surface, mintcream, position, radius,
linewidth)
>>> pygame.display.update()
```

Notes:



If you want to start again, just fill the surface with blue, and update again.

```
>>> surface.fill(deepblue)
>>> pygame.display.update()
```

Not responding

When you use PyGame interactively, like this, it can get stuck sometimes. This is because PyGame is trying to send events back to you – information about things it wants you to know about. At this stage, you don't need to know about them, but you should still tell Python to respond to them.

Type in this code and press Return twice, and you will see all the "events" that PyGame wants you to know about.

```
>>> for event in pygame.event.get(): print(event)
```

To close the PyGame window, use the close [x] at the top-right corner, as you would any desktop window.

Maths homework

Notes:

What a nightmare. Only just getting started and already there is homework. Computers and computer science share a great deal with mathematics. We've drawn a colourful circle or two, but now it is time to look at some number crunching.

To start with, let's go through a simple maths problem that you have probably been set at school at some time:

What are the factors of all the numbers from 1 to 50? That is, what are the positive divisors of the numbers from 1 to 50?

Without a Raspberry Pi

The first few are easy because you can divide the number by those numbers smaller than it in your head and see if there is a remainder.

1 -> 1
2 -> 1 and 2 divide into 2
3 -> 1 and 3
4 -> 1, 2 and 4. 3 doesn't divide into 4
5 -> 1 and 5 only. 2, 3 and 4 don't divide into 5 leaving no remainder
6 -> 1, 2, 3 and 6
7 -> 1 and 7 only
8 -> 1, 2, 4 and 8

Jumping ahead, let's try 26. We have to divide 26 by all the numbers smaller than it to be sure we have them all.

26/1 = 26
26/2 = 13
26/3 = 8 2/3
26/4 = 6 1/2
26/5 = 5 1/5
26/6 = 4 1/3
...
26/13 = 2
...
26/26 = 1

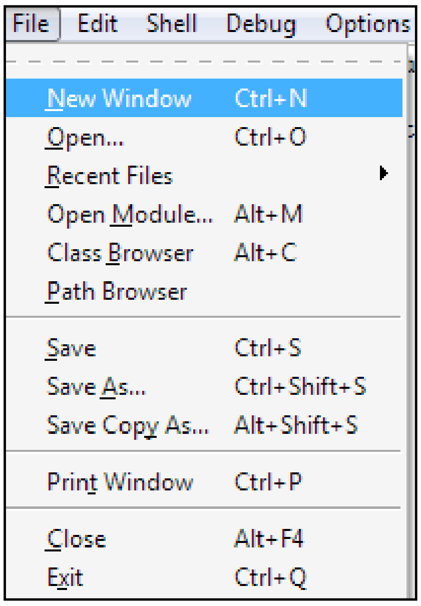
To work out larger numbers, you need to see which of those numbers smaller than it will have no remainder when they divide into it. In the case of 26: 1, 2, 13 and 26 leave no remainder.

Imagine that the question was just finding the factors of 12,407? You would have to start at 1 and work your way through more than 6,000 numbers to discover all of those that divide into 12,407 with no remainder.

Notes:

So, the same question. What are the factors of all the numbers from 1 to 50? Before you can start programming, you will need to start up an editor so that you can type in instructions to the Raspberry Pi.

Open a new editing window from the File menu in IDLE. This will allow you to work in the IDLE text editor.



Open up the IDLE editor as before, but this time use the **File** menu to select a new editing window. This is no longer the Python shell, but a text editor. You can type in your program but it won't be interpreted by your Raspberry Pi until you tell it to "Run". This text editor has been tweaked to make it very good for writing Python, just as a word processor has been tweaked to check your spelling and grammar.

In this window, type **exactly** what you see here, including the correct spaces before each line and all in lowercase. You can use Tab to indent your lines in the IDLE editor, and it will convert these to spaces.

```
for number in range(1, 51):
    print(number, ":", end=" ")
    for divisor in range(1, number+1):
        if number%divisor == 0:
            print(divisor, end=" ")
    print()
```

This program uses two loops, "range", "print", "if", "%" and "==", and two variables to hold data. The "range" function will generate the numbers from the first to one below the last; in this case, 1 to 50. The "%" operator will give the remainder after dividing the two variables.

Also, notice how "==" is used rather than just "=". This is because, on a computer, setting a variable to a number is different to comparing the equality of numbers. Using "==" makes it clear that you want to compare the numbers for equality.

The print function normally ends the line with a carriage return (the same as when you pressed the Return key). To stop this, you can tell the print function what you want at the end. In this program, we are telling "print" to put a space character at the end of the line instead. "print()" will end the line to give us a new row for the next answer.

Tip...

The indent of four spaces should appear automatically after the line ending with a colon (:). The indent of eight spaces will appear after the second line ending with a colon.

Tip...

In computer maths, the following symbols are used rather than what you might be used to:
** for multiply, instead of **x***
/ for divide, instead of ÷
*% for modulo, instead of / or **mod**.*

Notes:

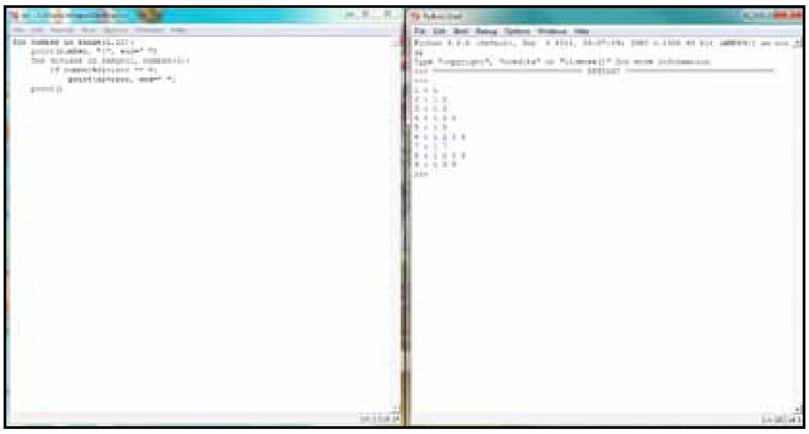
To make this program run, you can select “**Run Module**” from the editor’s Debug menu or simply press **F5** and it will ask you to save before you can run. Press “**OK**” and then give your first program a name – we will call it *“Factors.py”.*

NOTE: It is important to include the “.py” part of this name, so that Python can easily recognise this as a program it can interpret.

After you press “**Save**”, the program will run and you will see a lot of numbers display in the python shell. This is the result of your program, showing all the calculated factors.

```
>>> ============= RESTART ================
>>>
1 :  1
2 :  1   2
3 :  1   3
4 :  1   2   4
5 :  1   5
6 :  1   2   3   6
7 :  1   7
8 :  1   2   4   8
9 :  1   3   9
```

These are your factors for the first nine whole numbers.


The window on the right show the output of the Python code on the left.

Notes:

The Fibonacci Sequence

Let's write another program to work out a famous mathematical sequence. Each number in the **Fibonacci Sequence** is simply the previous two numbers added together; starting with 0 and 1. Create a new file using IDLE and name it "*Fibonacci.py*".

The following will calculate the numbers in the sequence up to the first value that is larger than 100.

```
first = 0
print(first, end=" ")
second = 1
while (first < 100):
    print(second, end=" ")
    third = first + second
    first = second
    second = third
```

And the result is:

```
0   1   1   2   3   5   8   13   21   34   55   89   144
```

This program should be easier to follow than the first. It has a single loop but, instead of using "for", it uses "while". This loop will continue while the variable "first" is less than 100. You should be able to see that "144" is displayed because it is the first value that passed this test. See what happens when you change "first < 100" to "second < 100".

The number 0

In almost all computing, number counting begins with 0. In the real world, it is common to count from 1. This can be very confusing until you get used to it – and even more confusing if you start using one of the few computer languages that does count from 1. The history of counting from 0 in computing stems from indexing into memory. You can say the first item in memory is a distance 0 from the start of its memory. The languages that are 1-based compensate for this automatically. Python is 0-based.

This can lead to even more confusion at the other end of your counted range. If you start at 0 and need 10 consecutive numbers, the last number will be 9. The "range()" function is not generating numbers from the first to the last number, it is starting at the first number and giving you the consecutive values after that. That is why in the factors example, the program appears to count from 1 to 51. Start at 1 and give me 51 consecutive numbers. Here is a way to iterate (i.e. count) through what the range function is generating.

```
>>> [ x for x in range(0, 10) ] # 10 numbers starting at 0
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> [ x for x in range(1, 51) ]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46,
47, 48, 49, 50]
```

Bits and bytes

Notes:

If you don't already know, computers work in "**binary**". Rather than counting from 0 to 9 for each unit as we do every day, they count from 0 to 1 for each unit. In decimal numbers, from the right, the numbers are units, then 10s, 100s, 1000s and so on. Each one is 10 times the previous one. In binary, the right-most numbers is a unit, then 2s, 4s, 8s, and so on. Each one is 2 times (double) the previous one.

Each unit in binary is called a "**bit**", and this can have one of two values: 0 or 1 (electrically, "on" or "off"; logically, "true" or "false"). When computers add up numbers, they need to have rows of bits to represent the numbers. By grouping 8 bits, we can make all the numbers from 0 to 255. This is known as a "**byte**". If we double to 16 bits we can have 65,536 different numbers (64k). You should start to recognise some of these computer numbers: 2, 4, 8, 16, 32, 64, 128, 256... Often, computer memory is sold in these sizes, and that is because they are made to hold bits of data.

Here are the decimal numbers 0 to 9 as they appear in binary:

Binary	Decimal
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
1001	9

Live Pythons

The previous experiments have thrown you in the deep end, using IDLE to enter programs and run them. IDLE requires you to have a graphical desktop (called a "**graphical user interface**", or "**GUI**"), but when you first started up your Raspberry Pi, it just shows you a lot of text on screen. This is called the "**terminal console display**".

In the infancy of the Linux operating system (the system used by your Raspberry Pi), GUIs were not common. Accessing a single computer was much like accessing the internet today: a screen and keyboard set on a console table with a wire into the back of a big box containing the hardware. When accessing your computer over a wire like this, you would not want to wait for images to download, so you simply had a text display. Watch any movie containing a computer older than 1990 and you'll see some examples. However, this is still the most efficient way to control a computer and many web servers (those things that send you web pages) are still maintained with a text interface.

So what has this to do with live Pythons? When you were typing lines of Python code to draw circles in PyGame at the start of this chapter, you were controlling the Raspberry Pi's graphical display much like web servers are controlled by their masters. Each command is interpreted line by line, and the computer is waiting for you to tell it what to do.

Notes:

Interpreting Python

Computers use instructions on the microprocessor to control them, and these are very fast but very complicated. Programming languages, such as Python, were invented to bridge the gap between how people think about problems and how a computer can process data. To keep the best performance, computers convert, or "**compile**", the human-readable instructions into those complex computational instructions before running them. Python isn't required to do that; although it can. By sacrificing speed, Python programs do not need to be compiled but can be interpreted line by line.

You can try this out for yourself. Open up a Linux terminal window from the menu on your desktop, you should be able to find "Accessories" and then "LXTerminal".

Alternatively, you can press Alt-F2 and type "LXTerminal".

Within the terminal window type:

```
$ python
```

You should see:

```
pi@raspberrypi:~$ python
    Python 3.2.2 (default, Sep 4 2011, 09:51:08) [RASPBERRY PI]
    Type "help", "copyright", "credits" or "license"
    for more information.
>>>
```

Notice how the prompt has changed from "$" to ">>>". This shows you that Python is active. When you want to get back to the Linux terminal, type "quit()" and press Return, but don't do that yet. When you see the "$" prompt, you have raw access to Linux. This is described in a later chapter of this manual.

Now, at the Python prompt, type:

```
>>> a = 0
>>> while (a < 10000): a+=1; print("Raspberry Pi Users = ", a)
...
```

Tip...

A semi-colon (;) is used to separate two commands without pressing Return.

It is not good practice to put all your code for a block into one line like this. You can press Return after each instruction here and Python will show you "..." to indicate that it is expecting more instructions but remember to indent each line in the block with spaces.

```
>>> a = 0
>>> while (a < 10000):
...     a+=1
...     print("Raspberry Pi Users = ", a)
...
```

Notes:

Tip...

When you see ellipsis (...) it means that Python is expecting more instructions to follow your "while" instruction. Just press Return again to clear it.

Running Python programs

Python programs don't have to run from IDLE either. You could use any application to create the text and then run the program using Python directly. You could even use a word processor but you'll find that it keeps trying to capitalise words at the start of each line and correct your grammar. It is **not** recommended you try to use a word processor for programming!

Those programs you created before can be edited using the "nano" text editor, for example. Open up the Linux terminal so you can see a "$" prompt. Open the program you created earlier – *factors.py* – in the nano editor, using:

```
$ nano factors.py
```

With some trial and error, you should be able to use nano to change the program to read:

```
for number in range(1, 51):
    factors = 0
    print(number, end=": ")
    for divisor in range(1, number+1):
        if number%divisor == 0:
            print(divisor, end=" ")
            factors+=1
    if (factors == 2):
        print("and is a prime number")
    else:
        print()
```

Save it ("Ctrl-O"), close nano ("Ctrl-X") and then, to run in the terminal window, type this:

```
$ python factors.py
```

You should see the program's output in the Linux terminal window, as it did in the IDLE Python Shell.

For the remainder of this Python section, we will use IDLE but, if you prefer, you can use a different text editor and run your programs from the Linux terminal as shown here.

Notes:

We've had a warm up, so let's see if we can create a game using Python. Have you ever played the Mastermind game, where you have to guess the hidden colours? We can do the same here, but with letters rather than coloured pegs. It will also give us the chance to work with some text in and text out commands.

Here's the code that we're going to use:

```
import string
import random

# create a list of 4 characters
secret = [random.choice('ABCDEF') for item in range(4)]

# tell the player what is going on
print("I've selected a 4-character secret code from the letters
A,B,C,D,E and F.")
print("I may have repeated some.")
print("Now, try and guess what I chose.")

yourguess = []
while list(yourguess) != secret:
    yourguess = input("Enter a 4-letter guess: e.g. ABCD : ").upper()
    if len(yourguess) != 4:
        continue

    # turn the guess into a list, like the secret
    print("You guessed ", yourguess)

    # create a list of tuples comparing the secret with the guess
    comparingList = zip(secret, yourguess)

    # create a list of the letters that match
    # (this will be 4 when the lists match exactly)
    correctList = [speg for speg, gpeg in comparingList if speg
== gpeg]

    # count each of the letters in the secret and the guess
    # and make a note of the fewest in each
    fewestLetters = [min(secret.count(j), yourguess.count(j)) for
j in 'ABCDEF']

    print("Number of correct letter is ", len(correctList))
    print("Number of unused letters is ", sum(fewestLetters) -
len(correctList))

print("YOU GOT THE ANSWER : ", secret)
```

Tip...

Here, again, you can see lines of code that don't fit on one line. Don't press Return to start a new line, just let it word wrap and Python will work it out for you.

Can you guess the number sequence set by the computer? The computer selects a sequence of four letters from a set of six. You enter a sequence of four letters that you think the computer selected and it will report how many were exactly right, and how many are in the sequence but are in the wrong position.

Notes:

This is a much longer program, which keeps asking for a new guess while the guess doesn't match the secret. The first thing it does is to randomly select a sequence of four letters from a choice of six. It builds this list sequence, which is denoted by "[" and "]", by looping four times and randomly choosing a letter from the options each time.

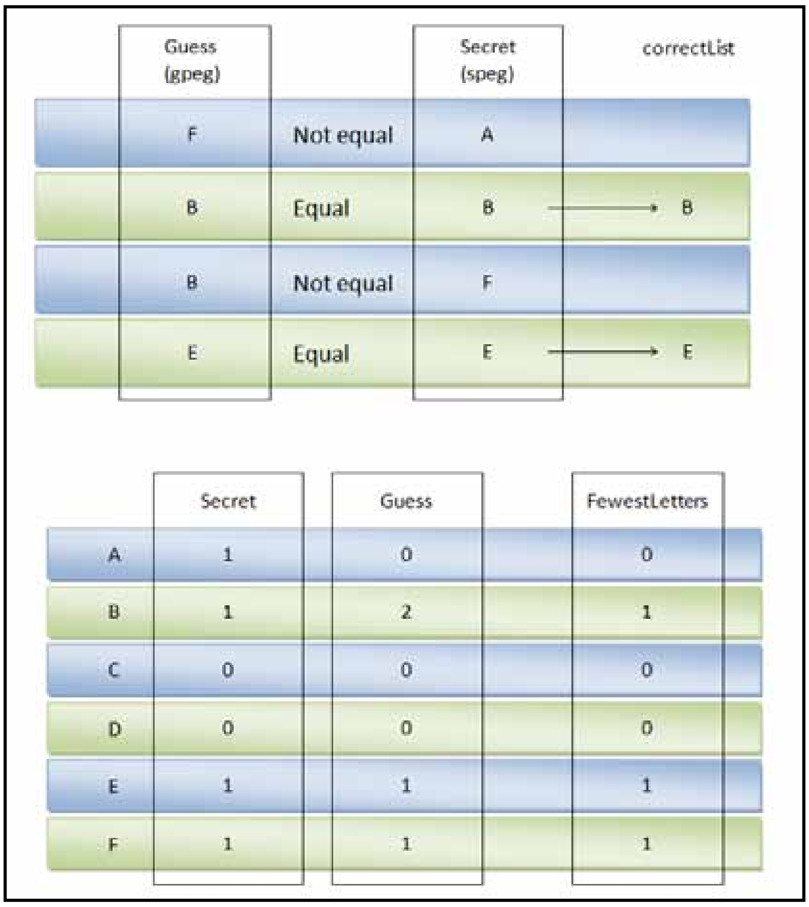
The computer gets the user to guess by waiting for input from them. Until the user types in something and presses enter the program will stop. The first thing it does with the input is to convert it to upper case (capital letters) and check it has at least four letters. The computer now needs to compare the secret answer with the guess. Next, it converts the sequence of letters from the input function to a list of letters, the same as the secret.

The next bit of Python magic is "zip". It takes the two lists and makes a new list, where each entry has two parts; a left part and a right part. This can be used to compare the left and right parts in a moment.

Now your program builds yet another list of letters that were correct; that is the left part and right part were the same in the comparing list. It creates the list by filling it with "speg" entries. "speg" means "secret peg", and "gpeg" means "guess peg". The program gets the "speg" from the comparing list along with the "gpeg", but it only uses the "speg" if it matches the "gpeg".

Correct Letters
= number in correctList
= 2.

Unused Letters
= fewest - correct
= 3 - 2
= 1



Guess (gpeg)		Secret (speg)	correctList
F	Not equal	A	
B	Equal	B	B
B	Not equal	F	
E	Equal	E	E

	Secret	Guess	FewestLetters
A	1	0	0
B	1	2	1
C	0	0	0
D	0	0	0
E	1	1	1
F	1	1	1

Notes:

The next bit of information that the player needs is how many letters were in their guess but not in the correct places. Since we don't care about the position, we are just counting letters. For each possible letter, we count how many are in the secret and how many are in the guess. This will include those that are in the correct position too, but we'll get rid of those counts in a minute. We also only want the minimum number since we are only counting those in the guess that are also in the secret. By adding up all these counts, and subtracting those that were correct, we have the number of letters that were in the wrong places.

This gives the player the clues they need to guess the answer.

This example is quite difficult but it is the ability to follow and understand this kind of program that will help you become a great programmer. The ability to create a logical process to solve a problem will help you on your journey to be a computing specialist.

Comments (# and """)

In the code, you will see the "#" character before some line of English. The "#" character starts a "comment" in the code. That is, everything that comes after the "#" is ignored by the computer. Python programmers use "#" to write little notes to themselves and anyone reading the code.

The art of writing useful comments is extremely important and a skill you should learn if you want to be a good programmer. After you have written a few programs without comments, left them for a month and then returned to them, you'll understand the value of good comments.

Comments can also be put inside a pair of triple quotes: """. You will see this in some of the listings. These comments, when placed at the start of a module, function or class, are known as "docstrings", which another computer program (PyDoc) can gather together to create a "user guide" to your program.

Notes:

```
def int2roman(number):
    numeralOrder = [1000,900,500,400,100,90,50,40,10,9,5,4,1]
    numeralLetters = {
        1000 : "M", 900 : "CM", 500 : "D", 400 : "CD",
        100 : "C", 90 : "XC", 50 : "L", 40 : "XL",
        10 : "X", 9 : "IX", 5 : "V", 4 : "IV",
        1 : "I" }
    result = ""
    if number < 1 or number > 4999:
        raise ValueError
    for value in numeralOrder:
        while number >= value:
            result += numeralLetters[value]
            number -= value
    return result

try:
    print(int2roman(int(input("Enter an integer (1 to 4999): "))))
except ValueError:
    print("Try again")
```

This program introduces the idea of a function. A function groups together commands into a block. It is the basic method by which you can create your own Python commands. Functions can have variables given to them and they can give back an answer once they are finished. They are great for reusing instructions on different data and getting results. In this case, it will take a normal decimal number and give back a sequence of letters that are the Roman equivalent.

The following command runs a number of functions all in one:

```
    print(int2roman(int(input("Enter an integer (1 to 4999): "))))
```

It prints the result of the function “int2roman”. It is started (**called**) with the result of “int” and “input”. The “int” function is converting the letters you type into a number. The “input” function allows Python to get some numbers from the user. Finally, The characters that the function creates are printed onto the display.

To work out the Roman Numeral equivalent for a number, you just need to keep taking the largest possible Roman number out of the given number, and adding the Roman Numeral to the string as it does so. A string is how computer scientists describe a sequence of characters; letters, numbers and other special symbols

Notice that the “numeralLetters” sequence is made from a pair of numbers and letters. This is a particular kind of sequence known as a “**dictionary**”. The order of the entries in the dictionary is not guaranteed to be the same as you enter it. This is so the computer can optimise how the data is stored and give you quick access later. The left part of the dictionary entry is the “**key**”, and can be almost anything. To get the value of the entry (the right-hand part), you can ask for it by the key, using square brackets “[ ]” around the key. The “numeralOrder” is a normal list, where the order is maintained. This is used to break down the number coming into the function, from highest to lowest.

Notes:

The main part of the Roman Numerals program also introduces the idea of "**exceptions**". When a function you write detects something wrong, you need to let the callers of your code know. You can do this by raising an error.

```
    if number < 1 or number > 4999:
        raise ValueError
```

"ValueError" is raised by other parts of Python when you attempt an operation on a variable that cannot be done, so we've used it here. The program catches the exception and prints a message.

```
except ValueError:
    print("Try again")
```

Troubleshooting those bugs

A "bug" is computer jargon for a mistake in your program. I have not intentionally put bugs in the listings, so if you find you cannot run a program or it gives a weird response, you have somehow introduced a bug. The term has been around since the time of Thomas Edison (c. 1878) but the often-reported case of a moth causing a problem inside a computer around 1947 cemented the term in the world of computers.

The ability to track down bugs is an invaluable skill for all programmers. With all the best intentions and planning, mistakes can be made. When starting to program, it is better to type in the examples listed here. It might take longer but you are in training. You are training to type and how to avoid making typing errors. Your ability to proof-read code will increase as you work through these examples.

When you find a bug, don't panic and start changing code at random. Look at the listing, look at your code, and work through each part logically to find out what you should be seeing, compared to what you are actually seeing. The errors given by Python will be confusing but it is only by following the instructions in your code and telling you why it cannot continue.

At some point, you will discover that computer programmers use another program called a debugger to analyse and find bugs in their code as it runs. You won't need the debugger just yet but there is an alternative method. The easiest method is to add "print()" commands to your code to find out what is going on in the variables and how far Python is through your program. Think of it as inserting a probe into the code to tell you what is happening.

When you have removed all the bugs and your program runs properly, you can then remove all those print commands to clean up the results.

Sorting and searching

Notes:

Let’s try something a little more practical, such as manipulating data and using files. For this experiment, you need to use a text window in IDLE to create a list of film names. Type the film names as shown here into the editor and save it as “*filmlist*”.

Inception (2010)
Source Code (2011)
Avatar (2009)
The Simpsons Movie (2007)
X-Men Origins: Wolverine (2009)
Rango (2011)
The Green Hornet (2011)
Grown Ups (2010)
Harry Potter and the Deathly Hallows: Part 1 (2010)
Harry Potter and the Deathly Hallows: Part 2 (2011)
Star Trek (2009)
Spider-Man 3 (2007)
Transformers (2007)
Shrek the Third (2007)
Kung Fu Panda (2008)
Mamma Mia! (2008)
Quantum of Solace (2008)
WALL-E (2008)
The Dark Knight (2008)
Up (2009)
The Twilight Saga: New Moon (2009)
Sherlock Holmes (2009)
Toy Story 3 (2010)
Despicable Me (2010)
How to Train Your Dragon (2010)

Now, let’s write some code to sort these films into order by release date.

```
# sort a file full of film names

# define a function that will return the year a film was made
# split the right side of the line at the first "("
def filmYear(film):
    return film.rsplit('(',1)[1]

# load the file into a list in Python memory
# and then close the file because the content is now in memory
with open("filmlist", "r") as file:
    filmlist = file.read().splitlines()

# sort by name using library function
filmlist.sort()

# sort by year using key to library function - the film list
# must end with a year in the format (NNNN)
filmlist.sort(key=filmYear)

# print the list
for film in filmlist:
    print(film)
```

Notes:

Instead of taking input from the user of the program, this experiment takes input from a file containing text. It processes the file to sort its lines into name and year of release order and then prints the list to the screen.

Getting the data from a file requires it to be opened, read into a Python list and then closed. There is a possibility that the file could be very large and not fit in the Python memory. We will cover how to solve this problem later.

Sorting the list is easy, as there is a function called "sort", which runs a well-established sorting algorithm on the list. It sorts by the initial letters (alphabetically) of each line. This is done first in this example. If you want to sort some other way, you need to provide the sorting **key**. This key is the bit of each list entry that is to be compared with other keys to decide which comes first.

The second sort uses a key for the film year. To get the year out of any list entry, the "filmYear" function splits the entry into two at the rightmost "(" and uses that. This is the year of the film, in our case. This program will fail if the film title does not have a year following the first final "(" in the name, or if there is a line in the file that contains no film name. Normally you will have to validate your input. One of the rules of programming for real people is: "Never trust your user".

Try removing each sort to see what happens. Try putting extra "(" in some of the film names.

Film search

```
# sort a file full of film names

# function to compare the lowercase item with the searchfor
def compare(item):
    return searchfor in item.lower()

# load the file into a list and
# close the file because the content is now in memory
with open("filmlist", "r") as film:
    filmlist = film.read().splitlines()

searchfor = input("Enter part of the film \
name you are searching for: ").lower().strip()

# use the built-in filter function, which will
# call the first parameter on every item on
# the list and, if it is true, it will use the item
foundlist = filter(compare, filmlist)

for name in foundlist:
    print(name)
```

Notes:

As in the previous experiment, the plan is to load a list of film names and then process them. In this case, the requirement is to find a film requested by the user.

The film list we created previously is loaded into memory. Text to “searchfor” is requested from the user. This is immediately converted to lowercase characters and any extra spaces stripped. During the search, any film title will also be converted to lowercase before checking so that case will not be relevant to the search.

There is a Python function called “filter”, which will take a list and generate a new list by filtering out any entries that do not pass a test. In this case, our test will compare the user’s query text with the entry being tested.

Lastly, the entries in the list of found items are printed to the screen.

Long lines

The input line has a “\” character half way through the question. This was added because the line was long. If the input question was typed on just one line, it would not need this character. A “\” character at the end of a line is a continuation marker. It indicates that the line hasn’t finished yet and it will continue on the next line of the program. When the program is run, this character won’t appear in the result – it is only used by the program interpreter to keep track of the flow of the code.

Notes:

Now we've learned to use functions, as well as manipulating data, let's get back to something a bit more visual. This time we're going to get on with displaying graphics, controlling widgets and using classes.

```python
# Include the system and graphical user interface modules
import sys
from PySide.QtGui import *

class MyWindow(QWidget):
    def __init__(self):
        super(MyWindow,self).__init__()

        # name the widget
        self.setWindowTitle('Window Events')

        # create a label (a bit of text)
        self.label = QLabel('Read Me', self)

        # create a button widget, position it and
        # connect a function when it is clicked
        button = QPushButton('Push Me', self)
        button.clicked.connect(self.buttonClicked)

        # create a vertically orientated layout box
        # for the other widgets
        layout = QVBoxLayout(self)
        layout.addWidget(button)
        layout.addWidget(self.label)

        # track the mouse
        self.setMouseTracking(True)

    def buttonClicked(self):
        """ Update the text when the button is clicked """
        self.label.setText("Clicked")

    def mouseMoveEvent(self, event):
        """
        Update the text when the (tracked) mouse moves over MyWindow
        """
        self.label.setText(str(event.x()) + "," + str(event.y()))

# start an application and create a widget for the window,
# giving it a name
application = QApplication(sys.argv)

# construct a widget called MyWindow
widget = MyWindow()

# show the widget
widget.show()

# start the application so it can do things
# with the widgets we've created
application.exec_()
```

Up to this point, the data being processed has been printed to the screen; but it could have easily been printed on paper or written out to a file. We've been putting text directly onto the console output. If you want to draw onto a window you will need to use a graphical window library. In this section we are using the "PySide" library (a version of the "Qt4" library), although another – called "Tk" – comes included with every version of Python.

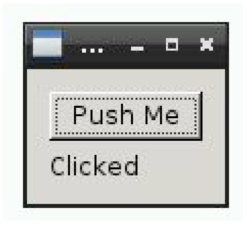
Notes:

When you tell a computer to change the image it's displaying, it does this by changing the colour of pixels on the computer screen. As with almost all computers today, the display on a Raspberry Pi is separated into what are called "windows". These windows "look" onto a drawable surface. Everything you see on a graphical computer display is actually a drawing within a window; even the drawing of text in your web browser.

A window must be controlled by an application (an app), so this is created first. The application must be running for the window to be displayed and, for this, it must hang around in a never-ending loop: the main loop. Before looping, you define how you want the window to look and how each part will react to stimulus (input). Finally you simply tell it to display.

To respond to external influences, such as mouse movement or keyboard key presses, you must override event functions. Some "**widgets**" (such as push buttons) can connect a function to a signal, such as "clicked".

In the experiment, the two methods for detecting changes are demonstrated. When the button is clicked, the "buttonClicked" function is called. The other is responding to the mouse cursor moving over the widget and changing the label to show the mouse position (but only when on the window, not when on the button).



Line drawing

Notes:

```
# Include the system, maths and graphical user interface modules
import sys, math
from PySide.QtGui import *
from PySide.QtCore import *

class SketchWindow(QWidget):
    def __init__(self, title, draw, size):
        super(SketchWindow, self).__init__()
        self.setWindowTitle(title)
        width, height = size
        self.setGeometry(60, 60, width, height)
        self.windowSize = size
        self.draw = draw

    def paintEvent(self, event):
        qp = QPainter()
        qp.begin(self)
        pen = QPen(Qt.yellow, 4)
        qp.setPen(pen)
        self.draw(qp, self.windowSize)
        qp.end()

# this draw function is not within the SketchWindow class
def drawSine(context, size):
    width, height = size
    points = []
    for x in range(width):
        angle = 5 * x * 2 * math.pi / width
        qpoint = QPoint(x, (height/2) * (1+math.sin(angle)) )
        points.append(qpoint)
    context.drawPolyline(points)

# create the application and widget and start it
application = QApplication(sys.argv)
widget = SketchWindow('Sine', drawSine, (320,200))
widget.show()
application.exec_()
```

Notes:

To draw a line, or a group of lines, first requires binding the paint event to a function. Hence, when the windowing system is ready to draw to the window it will call your function.

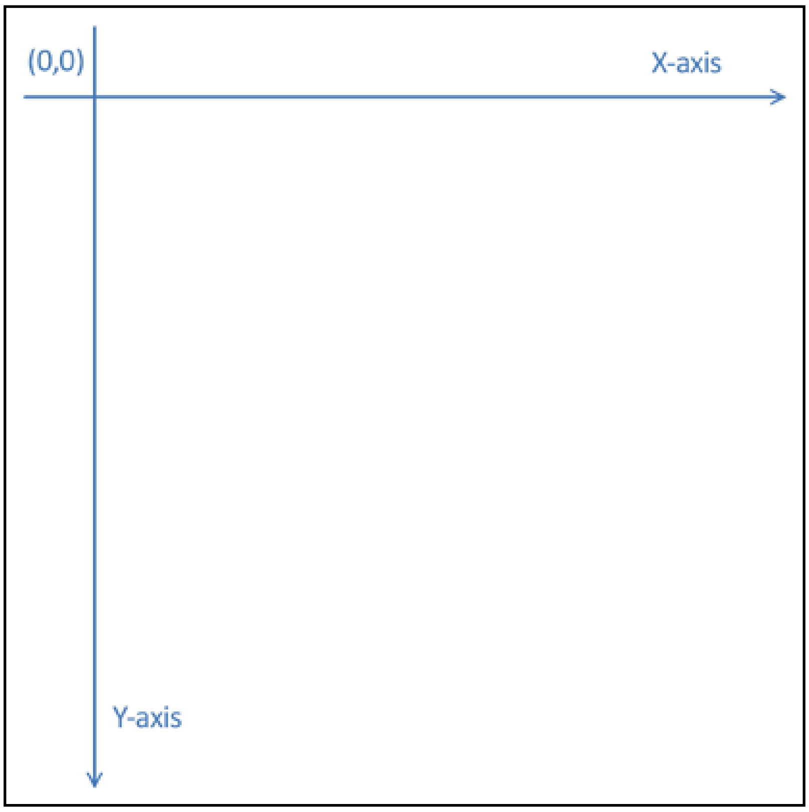
Before going to the main loop, we need to set up a QWidget within the app. This experiment and the last introduced the concept of a “**class**”. Understanding a class is quite complicated but think of it as a block of data that also defines the functions that can manipulate that data. You can create multiple blocks of data with the same class and they will all have the same functions being carried on different data. A program is basically a class with functions and classes within it.

The “SketchWindow” is a type of QWidget, and it will share all the functions had by a QWidget. The SketchWindow adds a new way to initialise (__init__) or setup the QWidget. It also defines a new way to draw the content of the widget. That is all we have to change, the normal function of the QWidget remains the same and so we don’t have to write that code too.

Notice how you’ve overridden the “paintEvent” with your own painting function. This paint function sets a yellow pen and then uses the drawing function that was given to the sketch window. The drawing function draws a sine curve five times, between the left edge and the right edge of the window frame. It does this by creating a list of points representing the curve. Sine can go from -1 to +1 but the calculation makes this range from 0 to 200. Finally, it joins up all the points using a “Polyline” to make a sine curve.

In this and the last experiment, you will have noticed that the top left of the screen and the content of our window is at coordinates 0, 0. When drawing on graph paper the origin (0,0) is at the bottom left. It is important to remember this when understanding how to define the coordinates of elements of your window.

Computers address the screen with the coordinate origin (0,0) at the top left, unlike drawing a graph.



Making a noise

Notes:

Graphics are all well and good, but if we are going to create true multimedia software then we're going to need to master audio output as well. Let's see if we can add some sounds to our programs.

```
import numpy
import wave
import pygame

# sample rate of a WAV file
SAMPLERATE = 44100 # Hz

def createSignal(frequency, duration):
    samples = int(duration*SAMPLERATE)
    period = SAMPLERATE / frequency # in sample points
    omega = numpy.pi * 2 / period

    xaxis = numpy.arange(samples, dtype=numpy.float) * omega
    yaxis = 16384 * numpy.sin(xaxis)
        # 16384 is maximum amplitude (volume)
    return yaxis.astype('int16').tostring()

def createWAVFile(filename, signal):
    file = wave.open(filename, 'wb')
    file.setparams((1, 2, SAMPLERATE, len(signal), 'NONE',
'noncompressed'))
    file.writeframes(signal)
    file.close()

def playWAVFile(filename):
    pygame.mixer.init()
    sound = pygame.mixer.Sound(filename)
    sound.play()

    # wait for sound to stop playing
    while pygame.mixer.get _ busy():
        pass

# main program starts here
filename = '/tmp/440hz.wav'
signal = createSignal(frequency=440, duration=4)
createWAVFile(filename, signal)
playWAVFile(filename)
```

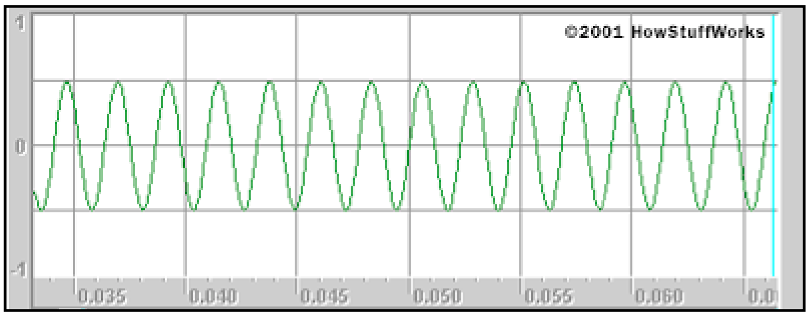
Playing sounds on a computer simply means taking data from a file, converting it to waveform data and then sending that to hardware that converts the waveform into an electrical signal, which then goes to an amplifier and loudspeaker.

On the Raspberry Pi, PyGame can be used to play sounds from a file. Rather than provide a file with some sound on it, we will generate a file with the data within it to play sound. You probably know that sound is actually the compression and rarefaction (stretching) of air to cause your eardrum to move.

That “action” is stored in a computer as numbers ranging from some large negative number to the same positive number, say 16,384 (2^{14} – remember your binary!), for a particular duration of time, say 4 seconds. If we move up and down that range evenly, say, 440 times each second, and pass those numbers to the sound hardware in the Raspberry Pi, we should hear the musical note “A”, above middle C.

Notes:

The sin() function returns a numeric value between -1 and +1, which represents the sine of the parameter entered between the brackets.



The evenness of this movement is given by the mathematic function we saw earlier, called sine, which when given an angle ranges from -1 to +1 for every 180 degrees. Given the full 360 degrees it goes from -1 to +1 and then back to -1 again. This gives the peaks and troughs. The “sin” function in Python doesn’t use degrees, it uses “radians”. There are 2π radians in 360 degrees.

We can’t just pass 440 peaks and troughs every second because we’d miss out all those values in-between. In the digital music world, CDs and MP3s send out 44,100 numbers to get the numbers in-between each second of sound. We can do that too, and this gives us the sampling rate of the music.

Holding a lot of data in Python can be inefficient, so some clever solutions for holding numbers can be found in a library called “numpy”. The first “arange” generates a long list of consecutive numbers – 44,100 for each second of music. Each is a point in time of each sample of sound. The “sin” function takes the list of numbers, adjusted by omega for the correct frequency, and gives a list of numbers representing the movement of the air for each point in time.

This list of sounds can then be saved to a file in a particular wave format. Finally, that sound wave can be played in the sound mixer in PyGame.

Normally, you wouldn’t generate sounds and then play them immediately. Sound files such as these would be created in advance, and only played within your own programs when required. However, if you were developing a sound-manipulation program, you would need to create and manipulate the sound data directly, a little like this. There are better methods of doing this that do not require writing to a file, which you might think about.

Lesson 3.5: Simulations and games

Notes:

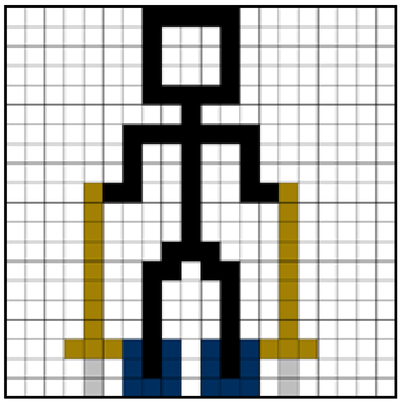
Now we can handle graphics and sounds, as well as manipulating data and using functions and classes. We can put all of this together to create something really worth seeing - it's time to write games!

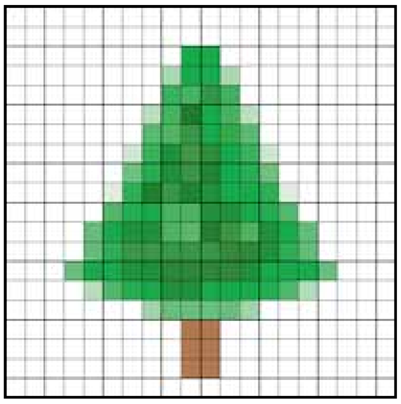
Let's go skiing!

For this experiment, you will need some little pictures ("sprites"), which will be moved around the display. You can use any "paint" software to create a picture of a skier, which is 20 pixels high and 20 pixels wide. Also, create a picture of a tree, which is also 20 pixels high and 20 pixels wide (see the pictures on the next page).

```
# pyGameSkier.py
import pygame

# a world sprite is a pygame sprite
class worldSprite(pygame.sprite.Sprite):
    def __init__(self, location, picture):
        pygame.sprite.Sprite.__init__(self)
        self.image = picture
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = location

    def draw(self,screen):
        screen.blit(self.image, self.rect)
            # draw this sprite on the screen (BLIT)

# a skier in the world knows how fast it is going
# and it can move horizontally
class worldSkier(worldSprite):
    speed = 5
    def update(self, direction):
        self.rect.left += direction * self.speed
        self.rect.left = max(0, min(380, self.rect.left))

class skiWorld(object):
    running = False
    keydir = 0

    def __init__(self):
        self.running = True
        self.skier = worldSkier([190, 50],
pygame.image.load("skier.png"))

    def updateWorld(self):
        # the skier is part of the world and needs updating
        self.skier.update(self.keydir)

    def drawWorld(self, canvas):
        canvas.fill([255, 250, 250])  # make a snowy white background
        world.skier.draw(canvas) # draw the player on the screen
```

Create these two sprites using some paint software. Save them as "skier.png" and "block.png" (the tree).





Notes:

Games are essentially real-time simulations of a system of rules that have a starting condition that is modified over time due to external input. Other than the external influences, the simulation is entirely predictable and deterministic.

Here is a simulation of ski slalom, where a skier makes his or her way through gaps in rows of trees. The game uses a random number generator to define where to put the gap in the trees. Rather than truly race the player down a slope, this game keeps the player still and races all the trees towards him or her.

This is a much more complicated system and the explanation of the simulation is in two parts. Firstly, how the program moves the skier on screen. Secondly, how to draw trees that the skier races passed.

Before the main loop is started, the game world needs to be created, or initialised. Game worlds are made up of objects and, in our case, these are the little sprites you created.

At the top of the program, a "worldSprite" class is created, which defines what a sprite is and what we can do to it. Then, we create a skier and enable it to move horizontally by a distance when it is updated. Lastly, we define the world, which is just a skier and some information about keys on the keyboard that have been pressed.

Even though we have defined these things, we haven't actually created them. This happens when we create the skiWorld.

```
def keyEvent(self, event):
    # event should be key event but we only move
    # if the key is pressed down and not released up
    self.keydir = (0 if event.type == pygame.KEYUP else
-1 if event.key == pygame.K_LEFT else
+1 if event.key == pygame.K_RIGHT else 0)

# pygame library wants to do a few things before we can use it
pygame.init()
pygame.display.set_caption("Ski Slalom")
pygame.key.set_repeat(100, 5)

# create something to draw on with a size of 400 wide
canvas = pygame.display.set_mode([400, 500])

# we will need to have a constant time between frames
clock = pygame.time.Clock()

world = skiWorld()

# check input, change the game world and display the new game world
while (world.running):

    # check external events (key presses, for instance)
    for event in pygame.event.get():
        if event.type == pygame.QUIT: # stop running the game
            world.running = False
        elif (hasattr(event, 'key')): # process this keyboard input
            world.keyEvent(event)

    # update the game world
    world.updateWorld()

    # draw the world on the canvas
    world.drawWorld(canvas)

    # important to have a constant time between each display flip.
    # in this case, wait until at least 1/30th second has passed
    clock.tick(30)

    # flip the display to show the newly drawn screen
    pygame.display.flip()

# once the game is not running, need to finish up neatly
pygame.quit()
```

Notes:

Adding trees

Notes:

```
import random

class worldTreeGroup(pygame.sprite.Group):
    speed = 5

    def __init__(self, picture):
        pygame.sprite.Group.__init__(self)
        treePicture = picture
        treeRow = pygame.sprite.Group()

        # in rows with a gap somewhere in the middle
        # only have a line of trees every 5th row or the
        # game is too difficult
        for y in range(0, 400):
            if (y % 5 == 0): # every 5th, add tree row with a gap
                gapsize = 3 + random.randint(0,6) # size of gap
                # starting position of gap
                gapstart = random.randint(0,10 - (gapsize//2))

                # create a row of 20 trees but 'gapsize'
                # skip trees in the middle
                for b in range(0,20):
                    if b >= gapstart and gapsize > 0:
                        gapsize-=1
                    else:
                        newtree=worldSprite([b*20, (y+10)*20],
treePicture)
                        treeRow.add(newtree)

            self.add(treeRow)

    def update(self):
        for treeRow in self:
            treeRow.rect.top-=self.speed
            if treeRow.rect.top <= -20:
                treeRow.kill() # remove this block from ALL groups
```

Now, if we can get rows of trees advancing on the skier's position, this game will be a lot more challenging. Also if the skier hits a tree, the game should stop.

To set up the "piste", this additional class sets up rows of trees that have a gap in the middle for the skier to pass through. The initialiser creates a group of trees by using the picture it is given and adding it a number of times (leaving a gap) to a group representing a row of trees. It then adds that group to the larger group representing a group of tree rows. Notice the use of integer division, "//", to divide the "gapsize". Without it, dividing gapsize by 2 may give a half and not a whole number (an **integer**).

The update part of the class defines what a group of trees can do when it is updated. The group will take all the rows of trees within it and move them up by a small distance. Also, when the row of trees goes behind the skier, off the top of the screen, it will remove that row of trees.

This world tree group does nothing yet because it has not been used by the main game. The next listing changes the main game to use the new class.

Notes:

skiWorld

Make these changes to the class “skiWorld”, don’t add them to the end of your program. See if you understand how to extend the skiWorld so that it will create trees, update them, draw them and check if the skier collides with the group of trees.

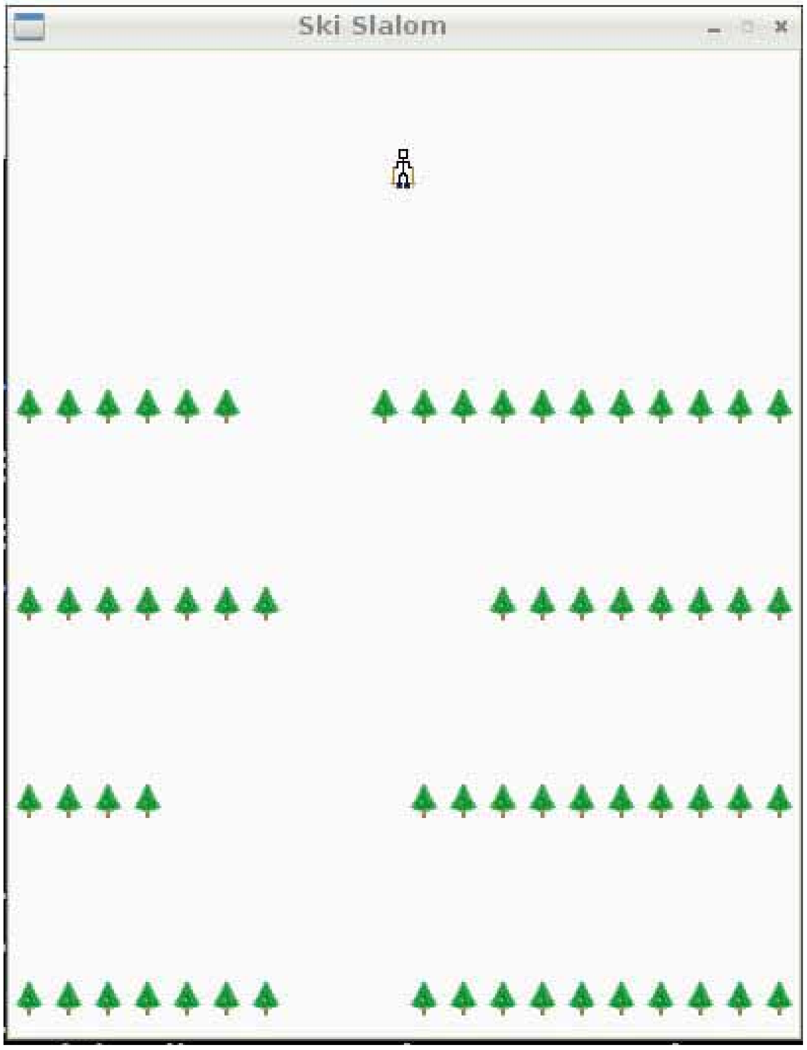
```python
def __init__(self):
    self.running = True
    self.skier = worldSkier([190, 50], pygame.image.load("skier.png"))

    ## adding trees
    self.trees = worldTreeGroup(pygame.image.load("block.png"))

    def updateWorld(self):
        # the skier is part of the world and needs updating
        self.skier.update(self.keydir)

        ## move the tree rows - removing any
        ## line that is off the top of the screen
        self.trees.update()

        ## check if the skier has collided with any
        ## tree sprite in the tree rows in the tree group
        if pygame.sprite.spritecollide(self.skier, self.trees, False):
            self.running = False

        ## check if the tree group has run out of tree rows -
        ## skier got to the bottom of the piste
        if len(self.trees)==0:
            self.running = False

    def drawWorld(self, canvas):
        canvas.fill([255, 250, 250]) # make a snowy white background
        world.trees.draw(canvas) # draw the trees
        world.skier.draw(canvas) # draw the player on the screen
```

In the skiWorld, one of these groups of trees is added when the world is initialised, the trees are updated when the world is updated and it checks for collision between the skier and any sprite in the group of tree rows. Lastly, if the tree group is empty, the game is over.

The only change to the main loop is that the group draws on the canvas.

You could try adding sound effects using the sound example earlier. If you can find some “wav” files from another source, you just need to play them. This will save you generating them yourself.

We now have our skier, our trees and our game. See if you can get him to the bottom of the slope!



Notes:

Notes:

```
import tempfile, heapq

# this is a generator (notice the yield)
def linesFromFile(sortedFile):
    while True:
        element = sortedFile.readline().strip()
        if not element:  # no element, so file is empty
            break
        yield element

# open the filmlist (this doesn't read anything yet)
films = open('filmlist', 'r')
generatorList = []
while True:
    # read up-to 10 lines into an array
    elementsToSort = []
    for i in range(10):
        film = films.readline() # just reads one line
        if not film:
            break
        elementsToSort.append(film)

    # if no lines were read, we're done reading the very large file
    if not elementsToSort:
        break

    # create a temporary file and sort the 10 items into that file
    sortedFile = tempfile.TemporaryFile()
    for item in sorted(elementsToSort):
        sortedFile.write(bytes(item.encode()))
    # return this file to the start ready for
    # reading back by the heapq
    sortedFile.seek(0)
    # put the generator into the generatorList array;
    # remember this function isn't executed
    # until a loop requests the next value.
    generatorList.append(linesFromFile(sortedFile))

# use the magical heapq merge, which will merge two sorted lists
# together but it will only pull the data in as it is needed
for element in heapq.merge(*generatorList):
    print(element.decode())
```

Notes:

The permanent storage on a computer (where all the data is kept when the machine is turned off) will usually be larger than the temporary storage (internal computer memory, which is lost when power goes off). On the Raspberry PI, your SD card (permanent storage) is likely to be much larger than the 256Mb of RAM (temporary storage). Look back at the previous experiment that loaded a text file, sorted the entries and then displayed the sorted list. If the file was larger than the fraction of 256Mb that was allotted to Python, how could we sort that file?

The Raspberry Pi's Linux operating system has a virtual memory solution but what if you wanted to keep down the Python memory being used? The answer is to sort part of the list, save that partially sorted list to the permanent storage, and repeat. Once the entire large file has been partially sorted to smaller files, merge all the entries in those files together one by one to make the final file. Very little is kept in memory.

The clever part here is to use a Python **generator** and **heap queue**.

Understanding a "generator" is quite difficult, but imagine that it is like a function that generates the elements of a sequence. After it "yields" an element of the sequence, it does nothing until it is called again. So the generator here takes a file, reads a line from it and yields that line to whatever called it. Once it cannot find any more lines, it just stops returning items, which the caller will notice and do something else. You've seen a generator already. It was called "range".

The file that is passed to this generator is a subset of the larger file that we want to sort. The first loop creates a lot of temporary files on your permanent storage, each of which contains 10 lines that have been sorted. The first loop doesn't make use of the sorted partial files immediately. It appends the generator to a list. It is making a list of the generator functions (linesFromFile) that will read lines one at a time from the file.

The "heapq" merge function loops through the values given by the generators and merges them in order. It assumes that the elements from the generator will come in a sorted order. So, imagine it takes the initial entries from each partial file until it has worked out which of these is the first. It gives that back, then it continues doing this one by one, only holding a small amount in memory. All that's left is to do something with each value as it is given. In this case, print it on the display.

Notes:

```
from urllib.request import urlopen
from xml.dom import minidom
import time

# extract weather details by grabbing tags from the RSS feed
# 'channel' contains all titles that contain weather heading text
def ExtractWeather(doc):
    for node in doc.getElementsByTagName('channel'):
        for title in node.getElementsByTagName('title'):
            print(title.firstChild.data)

results = []
bbc_weather = "http://open.live.bbc.co.uk/weather/feeds/en/"
locations = ["2653941", "2655603", "2633352", "2653822", "2650752"]
forecast = "/3dayforecast.rss"

start = time.time()

for location in locations:
    # open the rss feed and parse the result into a web document
    # and add the doc to the end of the list
    results.append(minidom.parse(urlopen(bbc_weather+location+
forecast)))

elapsedTime = (time.time() - start))

for webDoc in results:
    ExtractWeather(webDoc)

print("Elapsed Time: %s" % elapsedTime)
```

We're getting to the end of the experiments now, and these will require you to have a web connection. The first part of this experiment shows you how to get weather reports from an **RSS feed**. An RSS feed is like a trimmed-down webpage with only a standard set of data that can be grabbed and displayed. Whenever you look at a webpage or an RSS feed, you are really grabbing a list of instructions on how to draw the page in your web browser or feed reader.

This program is set up to find the three-day forecast for five different locations in the UK. It grabs the pages into what is known as a "**Document**" and then it extracts the information marked as "titles" from that document. These contain the forecast data on the RSS feed.

The main function for grabbing web data is "urlopen". A **URL** is a "**uniform resource locator**", which is the internet equivalent of a street address for anything and everything on the web. A call to "urlopen" will stop the Python program until it finds what you are looking for and returns a result. This might take less than a second but it could also take several seconds. This might be a problem, and it explains why you have to wait for your browser to respond to some webpages.

This program may take a few seconds before it completes fetching all five reports. The listing takes a snapshot of the time before calling "urlopen" and again after it completes so that it can show you how long it took.

Threading – why wait for responses?

Notes:

```python
import threading
from urllib.request import urlopen
from xml.dom import minidom
import time

# extract weather details by grabbing tags from the RSS feed
# 'channel' contains all titles that contain weather heading text
def ExtractWeather(doc):
    for node in doc.getElementsByTagName('channel'):
        for title in node.getElementsByTagName('title'):
            print(title.firstChild.data)

class urlOpenThread(threading.Thread):
    def __init__(self, host):
        threading.Thread.__init__(self)
        self.host = host

    def run(self):
        # open the rss feed and then parse the result into a
        # web document - add this to end of the results list
        global results
        results.append(minidom.parse(urlopen(self.host)))

bbc_weather = "http://open.live.bbc.co.uk/weather/feeds/en/"
locations = ["2653941", "2655603", "2633352", "2653822", "2650752"]
forecast = "/3dayforecast.rss"

results = []
startingThreads = threading.activeCount()
start = time.time()

# create a thread for each url open and start it running
for location in locations:
    urlOpenThread(bbc_weather+location+forecast).start()

while threading.activeCount() > startingThreads:
    pass

print("Elapsed Time: %s" % (time.time() - start))

for webDoc in results:
    ExtractWeather(webDoc)
```

When accessing the web, you will not get an immediate response to your requests. When you ran the previous Python experiment you may have been lucky and had it take little time to get your weather reports.

Computers can do many things very quickly and it is a shame to leave a computer waiting. This experiment shows how your computer can weave threads of instructions together so they appear to run at the same time.

To avoid waiting for all the pages to respond, you can tell the computer to run all five requests for these weather reports at the same time and not wait for others to return before storing the result.

The technique is called “**threading**”. Each thread runs at the same time as the others. Each weather report request is made using “urlopen”, as before, and each will stop its thread until it has a response but it will not stop the others. When each is finished it will continue. Threads can be used for anything that you want to execute at the same time but problems occur if they start changing the same bit of memory – so, **be warned**. Imagine writing an email on your computer and someone else also wanting to type into the same email at the same time!

The experiment defines a class that can open a URL as a thread and record the data returned in a global list of results. This class creates a new thread to open a URL for each web address, and starts it. The main program waits until the number of active threads returns to the same number as before the threads were started.

Notice how it takes much less time for the five URLs to be requested when they are executed concurrently.

Notes:

This is only the beginning – where do we go from here?

Notes:

These experiments may have left you exhausted and slightly confused, but I hope that they have also whetted your appetite to learn more about computer programming, computer science and computing in general. This is only the beginning.

This isn't a reference manual or a tutorial but the Python language comes with a comprehensive reference guide. If you don't have access to the internet, you can access the guide by typing:

```
$ pydoc -g &
```

Select "open browser" from the dialogue box that appears. This may take 20 seconds to appear, so be patient. From here, you can see all the Python documentation, PyGame and numpy.

If you do have access to the internet, there are plenty of resources covering how to program using Python. Now you have a skill for entering code, running code and hopefully toying with it, you can find out much more here:

- Beginner's reading
  http://wiki.python.org/moin/BeginnersGuide/NonProgrammers
- Learning with Python
  http://openbookproject.net/thinkcs/python/english2e/
- Learn Python The Hard Way
  http://learnpythonthehardway.org/book/
- Invent with Python
  http://inventwithpython.com/chapters/
- Python Tutorial
  http://docs.python.org/tutorial/

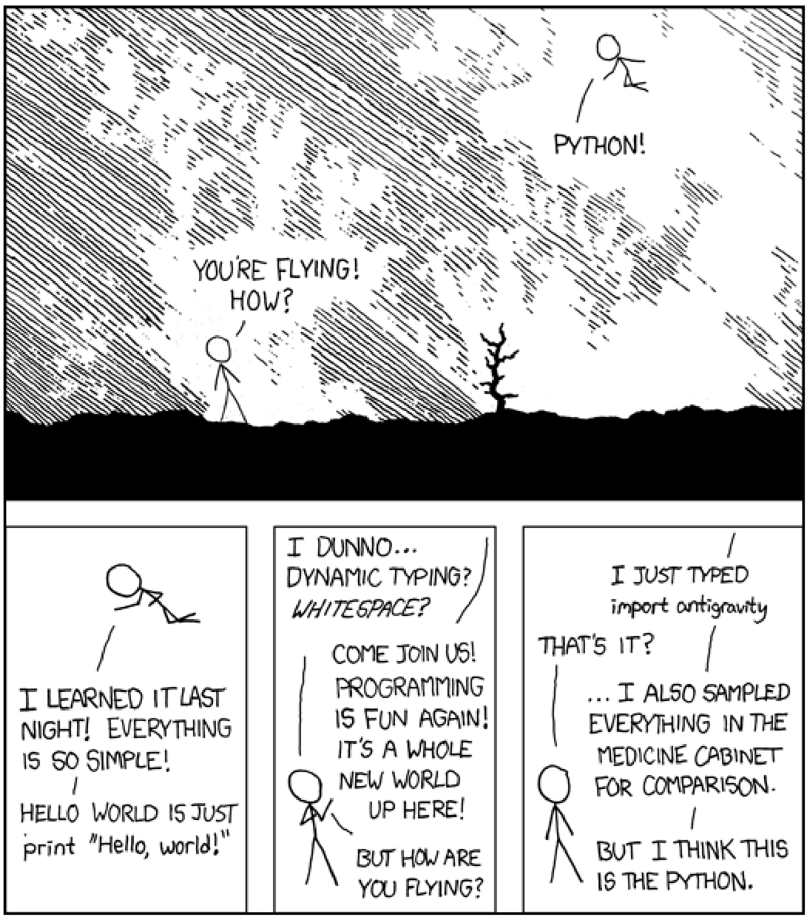
When you are stuck you can try the main source of all knowledge (for Python). For instance, information about str, chr, file, random, math, PySide, and heapq can be found at:

- Python Library Reference
  http://docs.python.org/library/
- PySide
  http://www.pyside.org/docs/pyside/
- PyGame
  http://www.pygame.org
- Python Package Index
  http://pypi.python.org

For information about print, while, yield, for, break, pass look at:

- Python Language Reference
  http://docs.python.org/reference/

XKCD #353 "Python"
http://xkcd.com


PYTHON!
YOU'RE FLYING! HOW?
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

Notes: