

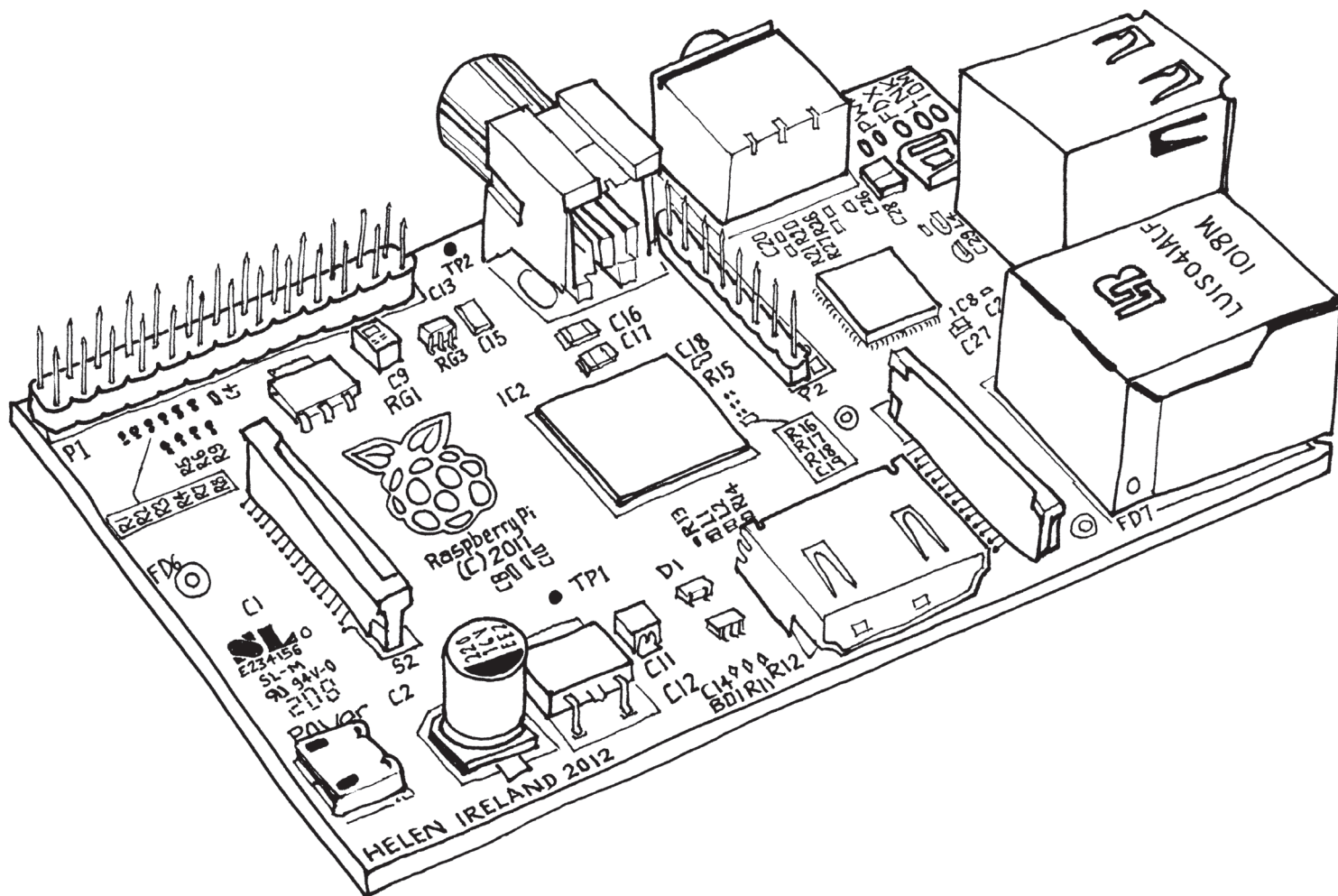
The Raspberry Pi **Education Manual**

COMPUTING AT SCHOOL
EDUCATE · ENGAGE · ENCOURAGE
In collaboration with BCS, The Chartered Institute for IT



The Raspberry Pi Education Manual

Version 1.0 December 2012



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.

000

0. Introduction 5

001

1. A beginner's guide to Scratch..... 7

1.1 Scratch basics 9

1.2 Moving sprites..... 15

1.3 Animation (loops)..... 18

1.4 Maths cat..... 23

1.5 Artificial intelligence 29

1.6 Control..... 35

1.7 Scratch games..... 44

What next? 50

010

2. Greenfoot on the Raspberry Pi.....

Coming soon!

011

3. Experiments in Python 72

3.1 Getting to grips with Python 73

3.2 MasterPy..... 86

3.3 Roman Numerals & data manipulation..... 89

3.4 Getting artistic..... 94

3.5 Simulations and games 100

3.6 Limited resources - memory & storage 106

3.7 Accessing the web - providing a weather forecast..... 108

This is only the beginning - where do we go from here? 111

100

4. Human-computer interfacing	113
4.1 Twitter	115
4.2 Email application	116
4.3 Remote Procedure Call	118
4.4 Web applications.....	120
4.5 General Purpose Input/Output (GPIO)	125

101

5. GeoGebra: fun with maths!.....	
Coming soon!	

110

6. The Linux Command Line	152
6.1 Commands are just programs	153
6.2 Command syntax and file structure	155
6.3 The superuser	161
6.4 Creating and destroying files and directories.....	163
6.5 Remote access to the Raspberry Pi	166

111

7. What next?	169
----------------------------	------------

Where are the Greenfoot and GeoGebra chapters?

The Greenfoot and GeoGebra chapters have been left out of this edition of the manual. These programs rely on software called a Java virtual machine, which is currently being optimised for the Raspberry Pi to improve performance. You can look forward to enjoying these chapters once we are happy that your user experience will be of the same high quality as the chapters themselves!

This manual was brought to you by...

This manual is a bit different. It was written entirely by unpaid volunteers, all of whom are keen to share their expertise and enthusiasm for computing with as many people as possible.

What all of these contributors have in common, apart from a youth spent mainly indoors in front of ZX Spectrums and Commodore 64s, is that they're all members of the organisation **Computing at School (CAS)**. To find out more about CAS and its work promoting the teaching of computer science, head over to <http://www.computingschool.org.uk>

Manual Contributors

Introduction by Andrew Hague
A beginner's guide to Scratch by Graham Hastings
Greenfoot on the Raspberry Pi by Michael Kölling
Experiments in Python by Andrew Hague
Human-computer interfacing by Ben Croston
GeoGebra: fun with maths! by Adrian Oldknow
The Linux Command Line by Brian Lockwood
Where next? by Clive Beale

Manual Production

Karl Wright, Robert Cruse and Paul Kingett of Publicis Blueprint

Digital Contributors

The following people offered contributions not covered in the manual, but available online and on your SD card.

Scratch Pong by Bruce Nightingale
Caesar Cipher by Brian Starkey
Fly by Alan Holt

Special Thanks

Martin Richards (University of Cambridge)
Simon Humphreys (Computing at Schools)
Alex Bradbury (University of Cambridge/Raspberry Pi Foundation)
Liz Upton (Raspberry Pi Foundation)
Eben Upton (Raspberry Pi Foundation)

Hello, Raspberry Pi users

Chapter 0

Congratulations! You have in your possession a Raspberry Pi. A small but powerful computer designed to help you understand and explore the almost-magical world of computing. Use it wisely; it's an object of great power.

What is the Raspberry Pi?

The Raspberry Pi is a computer, very like the computers with which you're already familiar. It uses a different kind of processor, so you can't install Microsoft Windows on it. But you can install several versions of the Linux operating system that look and feel very much like Windows. If you want to, you can use the Raspberry Pi to surf the internet, send an email or write a letter using a word processor. But you can also do so much more.

Easy to use but powerful, affordable and (as long as you're careful) difficult to break, the Raspberry Pi is the perfect tool for aspiring computer scientists. What do we mean by computer science? We mean learning how computers work so you can make them do what you want them to do, not what someone else thinks you should do with them.

And who do we mean by computer scientists? We mean you. You may finish this manual and decide you want to be next Tim Berners Lee, but even if you don't, we hope you have fun, learn something new and get a feel for how computers work. Because no matter what you do in life, computers are bound to be part of it.

Notes:

What am I going to learn?

This user manual is different. Don't expect a dry-as-dust description of how to plug things in or where to find your serial number. And you certainly won't learn how to create a spreadsheet or a presentation. That's really not computer science, it's something else entirely.

Instead, think of this manual, along with your Raspberry Pi, as a "computer science set". Have you ever been given a chemistry set? With a chemistry set, you can make lots of bangs, smells and odd-coloured goop to learn all about elements, molecules and compounds.

We're not going to make odd-coloured goop, but we will use experiments to discover how to program a computer to create your own games and animations, how to make graphics appear on screen just by typing in the right code (just like the developers of your favourite games do), how to get a cat to do your maths homework for you, and much more.

By doing all this, you will learn the basic principles of computer science. And that's your first step on the journey to becoming a real computer programmer, a games developer, an über-hacker just like in the movies (only cooler and staying strictly within the law) and many other things besides. Exactly what, depends on you.

Who is this manual for?

When we wrote this manual, our aim was for it to be suitable for most people of eight years and older. But that doesn't mean it's for eight year olds. This book is for anyone and everyone who is curious to know more about computing and creating computer programs. If you don't have computer-programming experience but you want to get some and you're looking for a place to start, this is it.

We begin the manual with some relatively easy experiments in computer science. Things then get progressively more challenging with each successive exercise. Try to spend time with each experiment and, once you've got an exercise doing what the manual says it should, feel free to change the code to see what happens: it's one of the best ways to learn.

Will I break it?

You can't break your Raspberry Pi by doing any of the experiments in this book, but you might just surprise yourself with what you can achieve. You will be working through and learning genuinely difficult but exciting concepts, and laying the foundations for even more exciting discoveries in the future.

So, without further delay, have everyone in the room stand back: we're going to do computer science!

A beginner's guide to Scratch

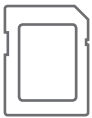
Chapter 1

Scratch is visual programming environment. With it, you can create your own animations, games and interactive art works. And, while you're doing that, you'll learn some important principles and techniques of programming without actually having to write your own code. It's a great way to get started. To find out more about Scratch, visit the web address ***scratch.mit.edu***

Notes:

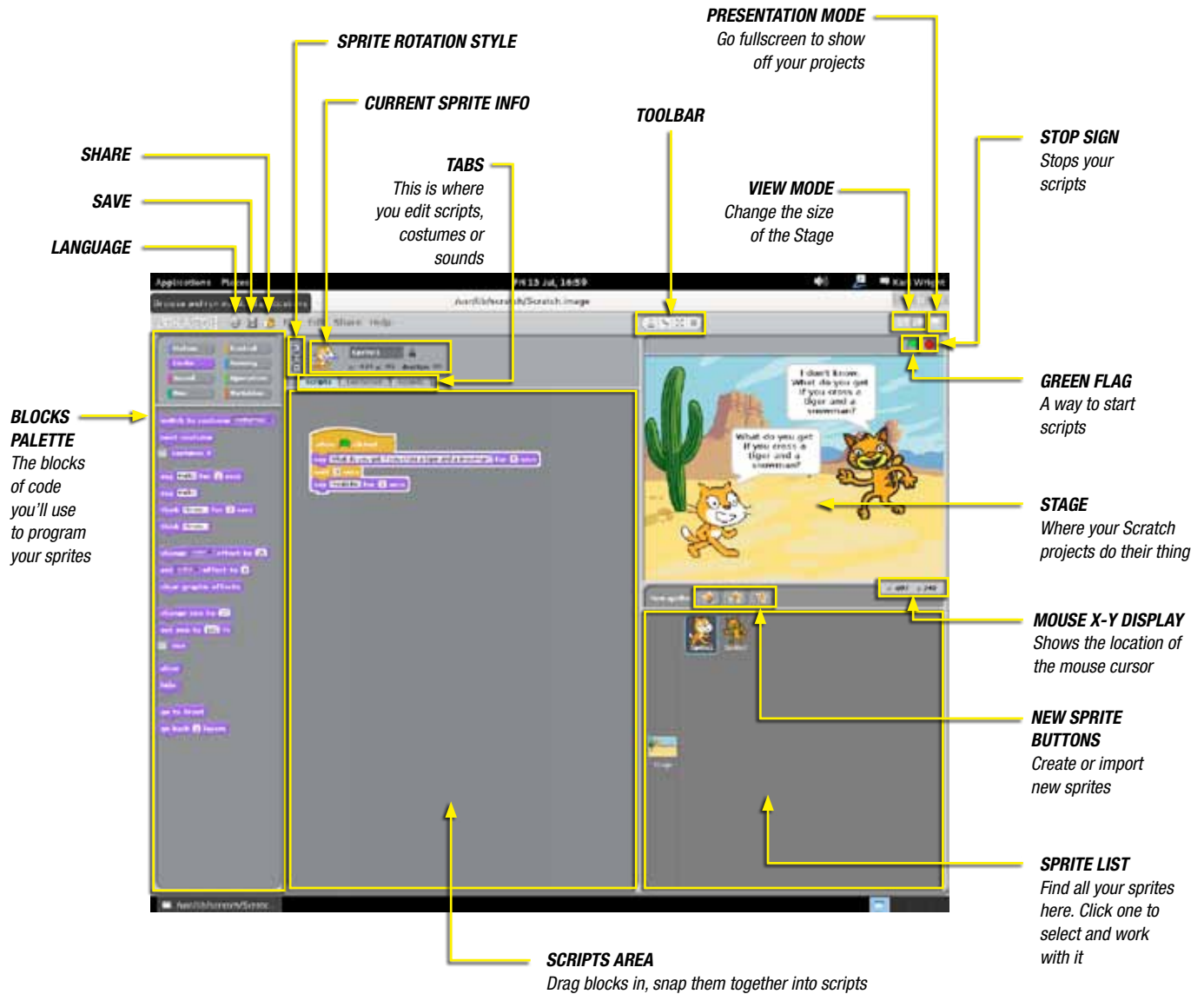
How to use this guide

We have tried to make this guide as straightforward to use as possible. To help you with the exercises in this chapter, we have already collected some little bits and pieces you will need, such as backgrounds, costumes for sprites, sound effects and complete examples of Scratch projects.



These can be found on the Raspberry Pi educational release SD card, in the folder /usr/share/scratch/RPiScratch. Wherever you see the SD card icon in the margin, that means we are referring to a file that can be found on your Raspberry Pi SD card. Go take a look! They can also be downloaded from Google Drive at ***<http://goo.gl/MpHUv>***

The Scratch interface



LEARNING OBJECTIVE: In this exercise, you will learn how to use the Scratch graphical user interface (GUI), how to create characters (sprites and costumes) and stages (backgrounds) for your projects, and how to add scripts.

RESOURCES: The sprites “cat” and “roman_cat”, and the background “roman_stage”.

Have you ever been in a school play? If you have, you’ll know that to put on a play you need a stage, actors, costumes and a script. Think of Scratch as being a bit like a play. The actors are called “**sprites**”.



You can dress your sprites in “**costumes**”, and each sprite can have more than one costume. The “**stage**” is the area on the screen in which your sprites will perform the tasks you write for them.



To make your sprites move and talk, you need to give them instructions. You do this by writing “**scripts**” using blocks of code from the Blocks Palette and Scripts tab on the left of the screen.

That’s enough introductions for now; let’s get to grips with the program itself.

Open Scratch from your Raspberry Pi’s Applications menu. You should now be looking at the Scratch graphical user interface, or GUI (pronounced “goeey”). Have a look around and tick the boxes below as you find these items:

- ☐ **1.** The stage (a big white screen)
- ☐ **2.** A sprite (clue: it’s a cat)
- ☐ **3.** The two costumes that your sprite can wear (click on the Costumes tab)
- ☐ **4.** The Scripts tab

Click on the Scripts tab, can you see any instructions for the cat to follow?

Let's have some fun with the cat

First, let's give the cat something to say. We'll start with "Hello, World". This is generally the first thing a computer programmer learns to do (don't ask me why). As you are now learning a programming language, you'd better start with "Hello, World", too.

Making the cat talk

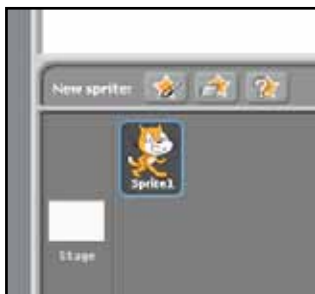
To make the cat say "Hello, World", we're going to be working with **"blocks"**. These are handy pieces of code, each containing an instruction for your sprite to follow.

There are eight different types of block. These can be found in the top-left corner of the Scratch GUI. They are colour-coded, so remember the colours. Find out what they are and complete their names in the table below:

M...	C...
L...	S...
S...	O...
P...	V...



Now, follow these simple steps to make your cat talk:



1 Click on the cat sprite in the Sprites List (bottom right) to make sure that it's selected.



2 Click on the "Looks" button in the Blocks Palette to make the Looks blocks appear.



3 Click on the block labelled "say [Hello] for [2] secs" and drag it to the Scripts tab.



4 Replace "Hello" with "Hello, World". Double-click the block and your cat should say: "Hello, World".

Notes:

We have to run a program to make it work. You can do this by just double-clicking your script, if you only have one script. But if we have more than one script, we might want to start them at the same time. We can use a “green-flag event” for this.

**TO FIND THE BLOCK FOR GREEN-FLAG EVENTS:**

1. Click on the Control button in the Blocks Palette.
2. Find the block labelled “when [picture of a green flag] clicked”.
3. Select it, then drag and drop it to the top of the script you’ve created in the Scripts tab. Make sure it snaps into place.

You are now ready to run your first Scratch program properly. Just click on the green flag symbol at the top-right-hand side of the Scratch window, just above the stage, and watch the cat do its thing.

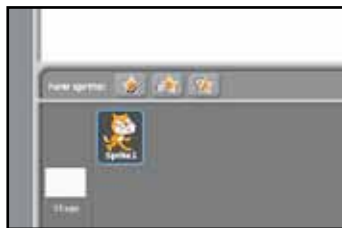
Over to you

QUESTION: For how long did the cat say “Hello, World”? _____ seconds

TASK: See if you can change the block to make the cat say “Hello, World” for 5 seconds.

Changing the way the sprite looks

Notes:



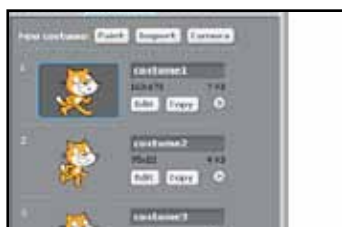
1 Click on your sprite to select it. In the Scripts area, click on the Costumes tab.



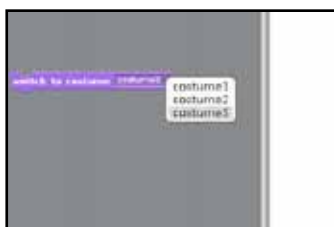
2 We are going to make a third costume for the cat, so click on Copy. A new cat costume should appear.



3 Select "costume3" and click on Edit. This will open the Paint Editor. Experiment with all the buttons and tools to find out what they do.



4 Once you feel at home, draw some clothes on the costume and click on OK. I gave my sprite a toga to make it look like a Roman Emperor.



5 Next, select the Scripts tab, click on the Looks button and select the "switch to costume []" block.



6 Drag it under the Scripts tab and use the drop-down menu to select "costume3". Double-click on this block and the cat will change his costume.

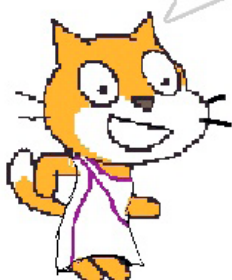
Now you have two blocks under the Scripts tab, one to say "Hello, World" and one for switching the costume. You can put them together by moving one so that it is just above or below the other. If a white line appears, the two blocks will snap together. Two or more blocks stuck together make a "script".

Over to you

QUESTION: Now that we have a script with two blocks, what happens when you double-click it?

TASK: See if you can arrange three blocks to make the cat change to his toga costume, say "Hello, World", then change back to its normal costume.

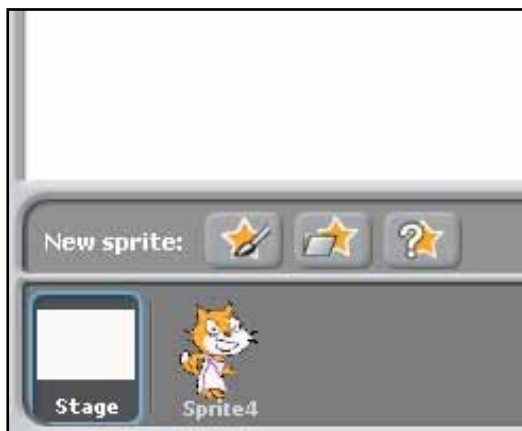
That cat's right: he looks like he's lost in a snow storm. We need to give him a stage on which to perform.



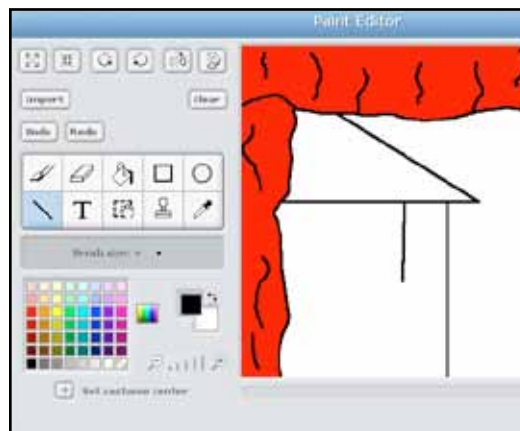
The stage

Notes:

It's time to give that cat a stage. We could be lazy and just import a picture to use as a background, but let's say that we're feeling energetic and want to draw our own.



1 Click on the stage in the Sprite List (bottom-right of the screen). Now click on the Backgrounds tab for the stage and click on the Edit button.



2 As before, the Paint Editor will open. Draw a stage for your sprite. When you have finished, click on OK. You can make further changes at any time by clicking on Edit.



3 Alternatively, you can import a ready-made background. Select Stage, then Backgrounds and then click on the Import button.

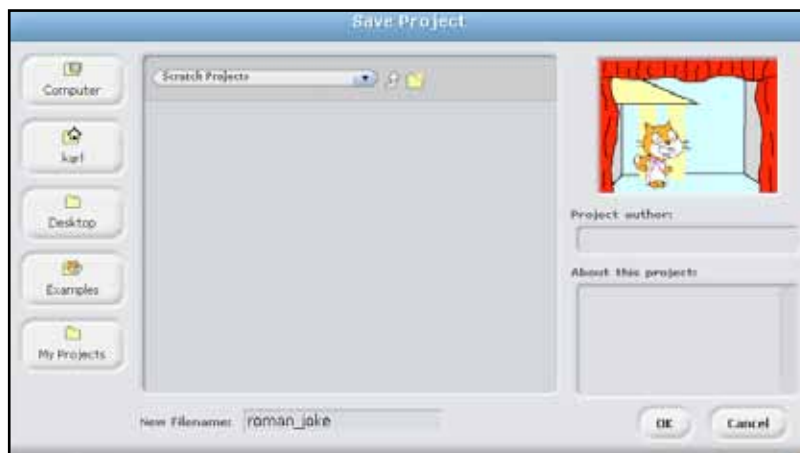


4 Have a look at all the available backgrounds before you pick the one that you want. We chose "roman_stage". Select the background by clicking on it with your mouse, then click on OK.

Saving your work

This is a good time to save your project. You would be wise to do this every 10 minutes or so, then you can be sure that you won't lose any of your hard work. When working on a big project, save it in two places, then you have a backup.

To save your project, click File, then Save – the Save Project window will open.



By default, it will save your work to the Scratch Projects folder. This is a sensible place to store your work, so type in a new filename, at the bottom. I've called mine "roman_play", so pick a different file name for your project or you will save yours over mine! Click on OK to save.

Wow! That is a lot for the first lesson. Have a play with Scratch – experiment with different blocks of code to find out what they do. Then come back when you have had a good rest and try Lesson 2.

Notes:



Tip...

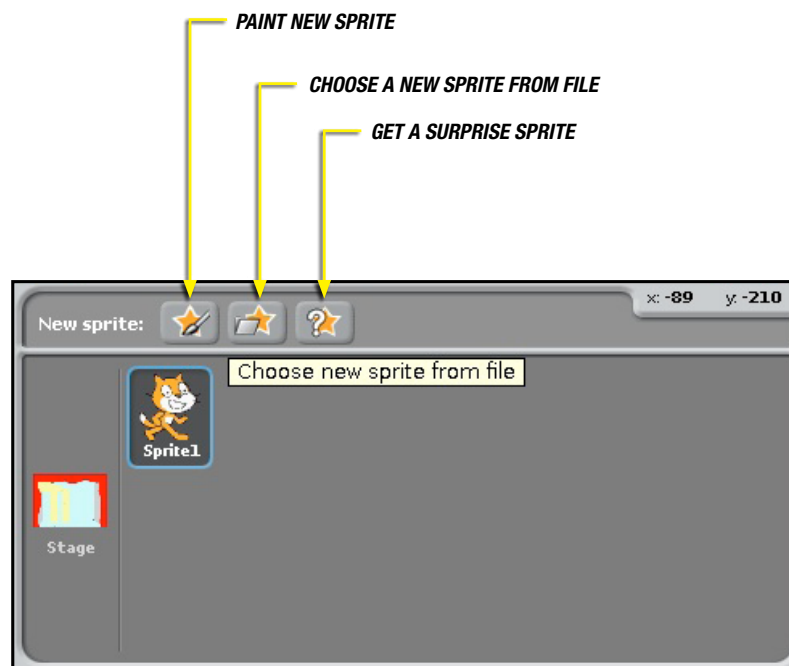
Use a name that will help you to find the project again. Always use an _ (underscore) between words in filenames – don't leave an empty space.



LEARNING OBJECTIVE: In this exercise, you will learn how to move sprites around the Scratch screen in a controlled way and how to tell a joke.

RESOURCES: The sprite "*roman_cat*" and the background "*roman_stage*".

The cat is feeling a bit lonely, so we'd better create some characters for it to play with. You can either paint your own sprites or import sprites from the Scratch Costumes folder. Use the New Sprites buttons to do this.



On the right-hand side of the program, just below the stage and above the Sprite List, you'll see three buttons: the New Sprite buttons. It's these we're going to use.

I want to add a time-travelling boy to my stage. To keep things simple, and to let us get on with some more programming, we're just going to import him.

Click on the middle New Sprite button and import the sprite "*boy4-walking-c*". But wait a sec: he's facing the wrong way! No problem. Go to the Costumes tab and click on Edit. Use the Flip Horizontally button to make him face to the left.

Use the Import Sprite button to find and import the sprite “boy4-walking-c”.



From the Costumes tab, click Edit and use the Flip Horizontally tool.



There are also buttons to make your sprite bigger, smaller, rotate counter-clockwise, rotate clockwise, as well as flip horizontally and flip vertically. Try them out. I have also used the shrink button to make my boy smaller.

Make your sprites tell a joke

Let's make the sprites tell each other a joke. You can do this using the speech block from the Looks category.

You could try a simple 'knock knock' joke to start with.

But wait! Are you finding that both of your sprites are talking at the same time.

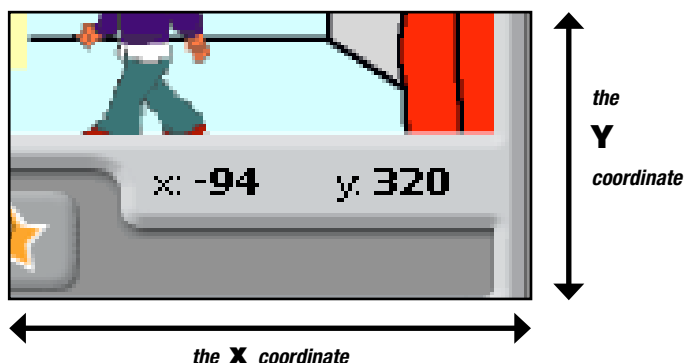
To fix this, from the Control block add the “wait [1] secs” block to the second sprite, before the “say” block.

Notes:

Positioning your sprite

Ok, we've told a joke. But this play is looking a bit static, so let's make our characters move. The first job is to move our two characters to their start points. In my play, the cat will come in from the left and the boy from the right.

You may have come across x and y axes when creating graphs.



The coordinates of any point on the stage are shown at its bottom-right-hand corner. Move your mouse around the screen and watch the numbers change.

TASK: Use your mouse to find the centre of the screen. Move the mouse pointer until it's exactly over the point x: 0 y: 0. Now let's position our sprites.

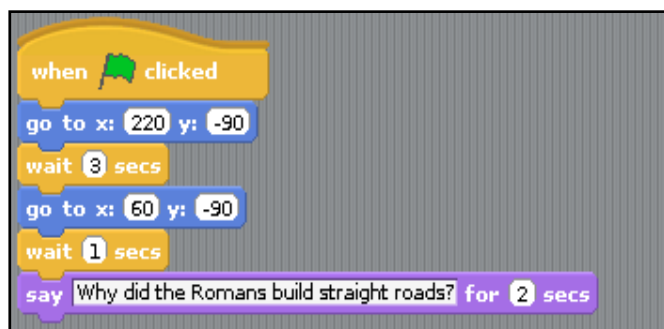
1. Select the cat sprite then, in the Blocks Palette, click on the Motion block labelled "go to x: [0] y: [0]".
2. Change the values in the block to x: -240 y: -80. This will take the cat to the far left of the stage.
3. Next place a "wait [1] secs" block into your script. This will give you time to see your cat before it moves.
4. Now add a second "go to x: [0] y: [0]" block. Use your mouse to work out the x coordinate just left of centre on the stage, to which we want to move the cat.

Repeat this process for your other sprite, positioning it slightly to the right of centre stage. Ideally, the two sprites should move from the edges of the screen to stand face to face, separated by a small gap.

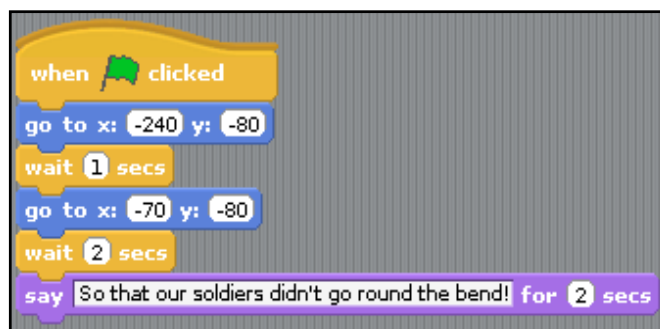
Now you need to make the sprites tell a joke. Remember to leave a short delay after each sprite speaks, otherwise they'll talk over each other. Have a look at the screenshots to see our code (and our fantastic joke).



Use your mouse pointer to find the coordinates of a position on the stage and make a note of those coordinates on a piece of paper.



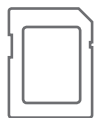
This is what our code for the boy sprite looks like.



And here's our code for the cat sprite. Does your looks the same?

TASK: Now add some code to your other character to move it to the right of the stage and then after a short delay move it into the centre stage.

Well done! You have certainly got the hang of moving sprites about the screen. Why not add some more characters to your stage and get them to tell jokes?



If you are having problems, you can load the sample code, "roman_play.sb", to see how the program is put together. Feel free to change things and to experiment, as this is a great way to learn.

Lesson 1.3: Animation (loops)

LEARNING OBJECTIVE: In this exercise, you will learn how to use repeat loops to create simple animations.

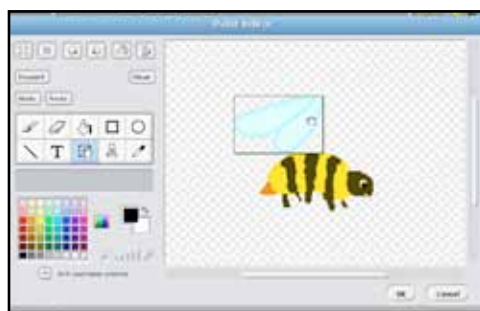
RESOURCES: The sprites "bee", "female_flower" and "male_flower", and the background "flower_bed".

With its animated characters, Scratch is great for telling stories. I have to do a school science project on pollination, so I have decided to use Scratch to tell the story of pollination in moving pictures. You can help me by following these instructions to animate a bee in flight.

First, open the file "bee1" from the "Animals" folder in the Scratch gallery. Next, import the background "flower_bed", this time from the "Nature" folder in the Scratch gallery. Delete the cat sprite; we don't need it for this project.

Copy "bee1", then edit "bee2" using the Select and Flip Horizontally tools, to make its wings point downwards. Together, the two costumes – "bee1" and "bee2" – will become an animation of a flying bee.

Copy your bee, then edit "bee2" so that its wings point downwards.



We need some script to make the bee look as if it is flying. We do this by switching from one costume to another and back again, making the bee appear to flap its wings. As we do this we will also make the bee move forwards.

Now, build your own script to make the bee fly. You will need blocks from Control, Looks and Motion. If you get stuck, have a look at the screenshot of our code. You'll find it further on in the lesson.

This is the code to make your bee fly. Instead of using the green flag to run my code, I will use a "when Sprite1 clicked" block from Control. The code will run when I click on the bee.



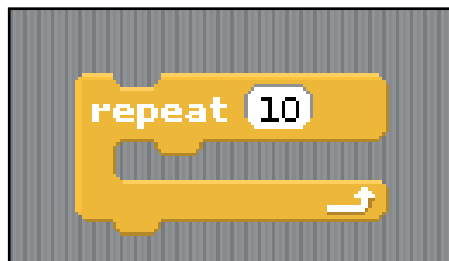
Here are the steps you need to follow:

1. Start with costume "bee1".
2. Add a "wait [0.2] secs" block, so that the viewer has time to see the costume.
3. Move the bee on 10 steps, before switching to costume "bee2".
4. Add another "wait [0.2] secs" block, so that the viewer has time to see the second costume.
5. Move the bee on another 10 steps.

But we need to do this more than once. To make the bee fly across the screen, we might have to repeat this 20 times.

Don't panic! You are using a computer. Computers are fantastic at doing things over and over again. They can do this very accurately and never get bored, tired or fed up.

What we need is a repeat loop.

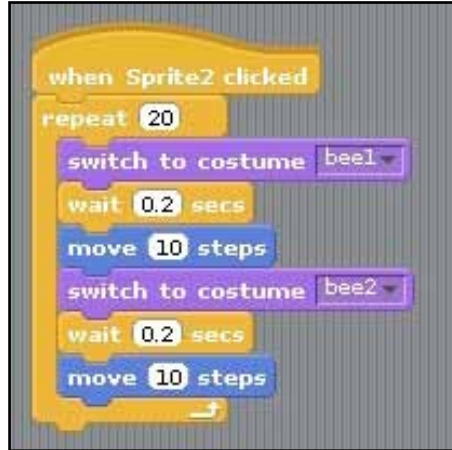


This is what we use to program the computer to repeat something over and over again. You will find the repeat loop ("repeat [10]") in the Control blocks.

It looks a bit different because it has a gap so that we can put code you want to repeat inside it. Just drag the “repeat [10]” block to sit directly under the “when Sprite1 clicked” block. It will automatically fit around your block of code, causing it to repeat itself.

Here is the code we need to animate the bee so that it flies all the way from one side of the screen to the other.

Using the repeat loop, you can make the sprite do the same actions over and over again. So the bee flaps its wings up and down many times.



Notes:

Over to you

QUESTION: Why do you think I have increased the number of times it repeats from 10 to 20?

TASK: Some of the Scratch sprites already have two costumes. Check out the Scratch cat sprite. Use its two costumes and code similar to the example on the left to make it walk.



Tip...

If your bee flies in the wrong direction, check the sprite to make sure that it is facing in the correct direction. With the sprite selected, look at the bar above the Scripts tab. The “forward” direction of a sprite is indicated by a little blue line. The bee on the left will move 90° to the vertical and the bee on the right will move 60° to the vertical. You can rotate the line to change the move direction of a sprite.

This is the storyboard of my pollination project. I added two more costumes to my bee sprite to show it carrying pollen and I have drawn a flower sprite with some stamens in blue.

But I didn't want to stop there. I wanted the bee to visit the second flower – a female – from the other side of the screen, so I copied all four costumes for the bee sprite and flipped them horizontally.

I also copied, flipped and edited the male flower to create a female flower sprite. I have given it two costumes. One shows the stigma without pollen and the other with pollen. Let's have a look at the resulting animation.



1. A bee flies towards a male flower.



2. The bee pauses to suck up nectar and collect pollen.



3. The bee then flies off with pollen from the flower's stamens.



4. The bee flies toward the female flower.



5. The bee sucks nectar and this time deposits pollen.



6. The bee flies off leaving pollen on the stigma.

I also decided that I only wanted one flower on screen at a time. So, I had to add scripts to make my flowers disappear and appear at the right points in the animation. I used the Looks blocks "show" and "hide" for this.

Here's the code for all three sprites in the pollination project.

Notes:

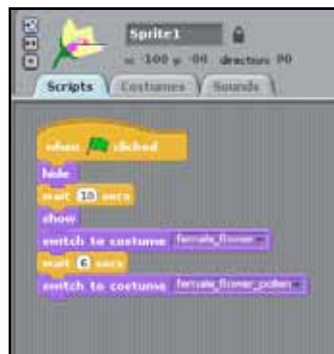
Code for the bee



Code for the male flower



Code for the female flower



Because I had three different sprites with their own scripts, I used the "when [green flag] clicked" event to run them all together.

Wow! That is quite a complicated project, but if you break up your animation into lots of little scenes it makes it easier to plan and to program.



To see what the whole project looks like once it's finished, open RPiScratch/Projects/ pollination.

Tip...

Import existing sprites and a background from the Scratch picture folders. This can save you a lot of time.

LEARNING OBJECTIVE: You will learn how to use variables to store data for using in a program. You will also learn how to use operators to do simple sums.

RESOURCES: The default sprite “cat” on the default white background.



Do you find maths difficult?

Can you imagine what it would be like to be able to do millions of sums in seconds and always get them right? Even the most complicated sums you can think of?

Computers are fantastic at maths. In fact, maths is what they do best. We can program the Scratch cat to do maths. The cat will ask for some numbers and then do the sums. So, how are we going to put numbers into the program for the cat to use?

When we input numbers (put numbers into a computer), the computer has to have somewhere to store them. Different people might input different numbers, so these numbers are going to be different each time.

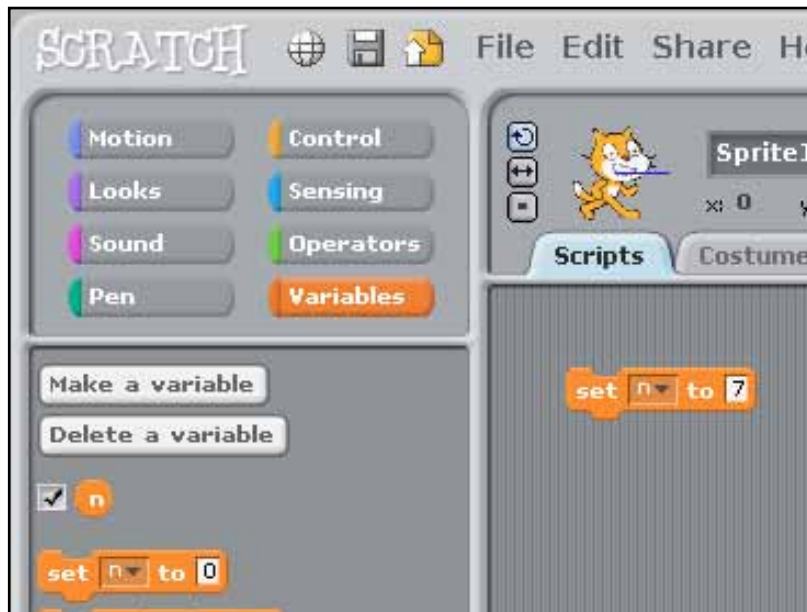
When programming, we store numbers in something called a “**variable**”. One way of thinking about a variable is as a box, or container, in which we can store numbers, letters or words.

We may have more than one variable in a program, so we give them different names. The name can be as simple as a single letter (or as complicated as you like!). For example, if it is storing a number, we might call the variable “n”.



In the diagram, we have stored the number “7” in the variable “n”. So we can now say “n = 7”.

In the example above, we created a variable called “n” and stored the number “7” in it. In Scratch, you would do this in two steps: first creating the variable “n”, and then using a block from Variables to set its value to “7”.



Notes:

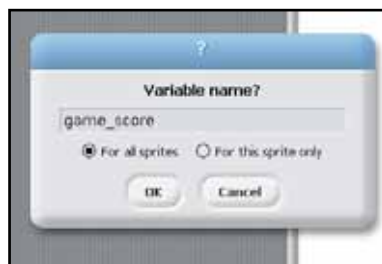
Tip...

Give each variable a name that reminds you what is stored in it. For example, if you are creating a game and you use a variable to store the score, then a good name for the variable would be "game_score".

If you wanted to use a two-word name for your variable, you would separate the words with an underscore (the "_" character), not a blank space.

Using variables in Scratch

Ok, now we're going to create and use variables. Click on Variables in the Blocks Palette and create a variable called "game_score".



1 Click on the Variables button, then on the button labelled "Make a variable". This opens the "Variable name?" dialogue window. Enter the name "game_score" for your variable and click OK.



2 Drag "Set [game_score] to [0]" to the Scripts tab. Then, from Control, drag the block "when Sprite1 clicked". Join the two together to make a script.



3 The default value for new variables is "0". Select the block "set [game_score] to [0]" and change the value to "100".



4 Finally, from Looks, I have used a "say [Hello] for [2] secs" block but changed it to "say [Great, I've got 100 points] for [2] secs".

In this example, we set the cat's score to 100 points by clicking on the cat sprite. But there are plenty of other ways to put a number into a variable. The method you choose will depend on how you want to use the variable.

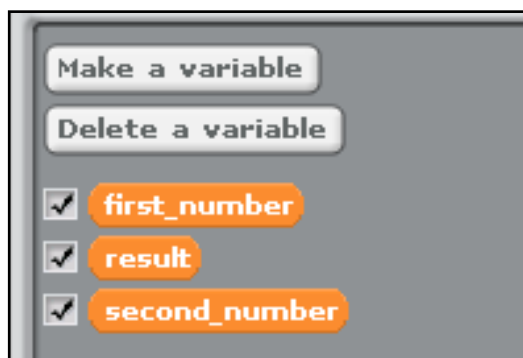
Notes:

Inputting the numbers

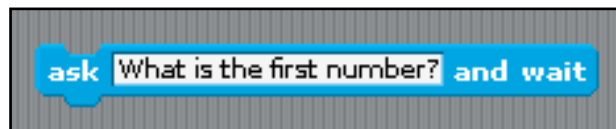
Your teacher has set you four sums. The sums are:

$56 + 39 =$
 $87 - 42 =$
 $16 \times 9 =$
 $240 \div 6 =$

We're going to show you how to get Maths Cat to do this homework for you. Let's start with the first question, $56 + 39$. If you get stuck, refer to the image of the completed code at the end of this exercise.

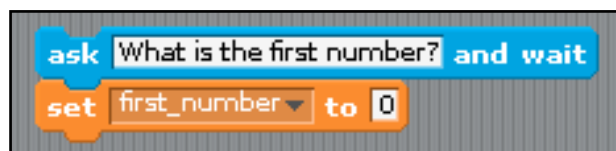


First, create three variables, and name the first two "first_number" and "second_number". You'll use these to store your two numbers. Name the third variable "result". This is where you'll store the answer to your sums.

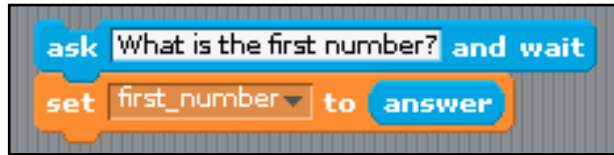


To input the first number you need to use a Sensing block to tell the user what to do. Drag the block "ask [What's your name?] and wait" to the Scripts tab. Change the value to "What's the first number?".

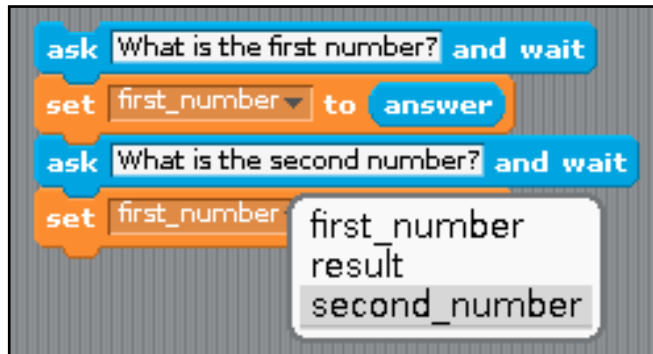
The answer given by the user is then entered to a variable, where it's stored for use in your sum, by using a combination of a set "variable" block from Variables and an "answer" block from Sensing. Let's see how.



Click on Variables, select the block "set [first_number] to [0]", drag it to the Scripts tab and snap it to the previous block.



Click on Sensing, select the block labelled “answer” and drag it onto the number “0” in the previous block. Your script should look like the one in the screenshot above.



Now the program knows the first number in your sum. To tell it the second number, repeat the process above but remember to change “first_number” to “second_number” in the drop-down box of the variable block.

Now for the really clever bit

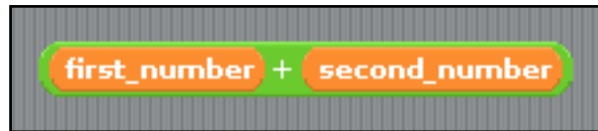
To do sums in Scratch, you need something called an “**operator**”. That’s just a fancy term for maths signs such as +, -, × and ÷. Yes, you guessed, we find these under the green Operators category in the Blocks Palette.

*Click Operators:
the “add” operator
 (“[] + []”) is the
first one on
the list in the
Blocks Palette.*



We’re going to use the “add” operator, so drag it to the Scripts tab. We’ll use it to add the variables “first_number” and “second_number”.

From Variables, grab the variable “first_number” and drop it into the first blank space in the “add” operator. Drop “second_number” into the second blank space (see the screenshot on the next page).



We're not finished yet. We've told the program to add our numbers, but we need to store that answer in one of our variables.

From Variables in the Blocks Palette, drag "set [first_number] to [0]" to the Scripts tab. Change "first_number" to "result" and drag your "add" operator onto the number "0". Your block should look like this:



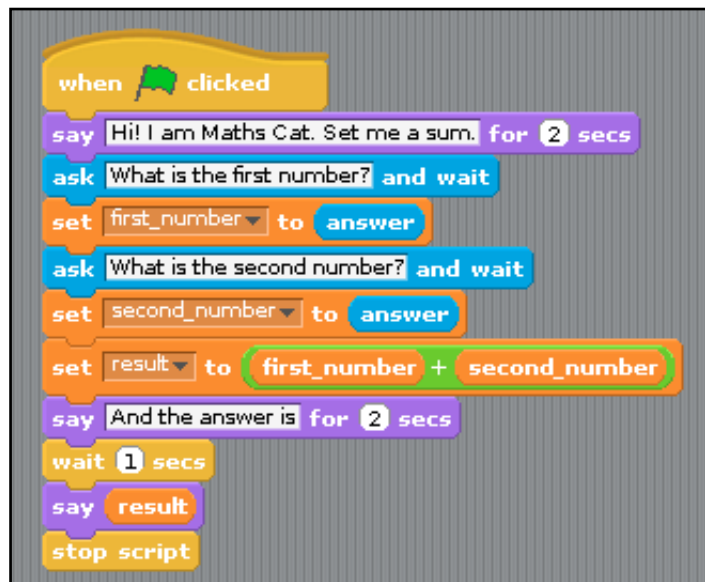
Finally, we also need to tell Scratch to display the answer to the sum. Otherwise, it will keep it to itself (and that's no good to us).

Click on Looks in the Blocks Palette. Select "say [Hello!]" and drag it on to the Scripts tab. From Variables, drag the variable "result" and drop it onto "Hello!". The image below will show you what we mean.



And that's it! I have included the whole program below. We've added in some extra blocks, to make the cat a bit chattier but the basics are the same as the script we built above.

Use this screen-shot to build the whole script. And remember, each time you want the cat to do another sum you will have to click on the green flag.



But hang on a minute: if you tried to make the cat do all the homework you will have noticed a big problem.

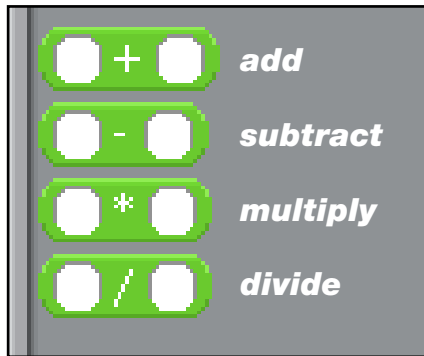
The cat can only do addition! Don't panic – a quick edit to change the program slightly will do the trick.

Tip...

Make sure all the blocks snap into place properly, especially when you have to snap blocks on top of other blocks.

Click on Operators once more in the Script block and you'll see that there are other operator blocks there too: for subtraction, multiplication and division.

Notes:



If you want the cat to take away then simply swap the “add” block in the script above for a “subtract” block, and so on until the cat has done all your homework for you.

What a helpful cat!



Over to you

TASK: Program the Maths Cat to do the rest of the homework sheet – you will have to change the operator each time so that the cat does the right kind of sum.



To see a working example of this script, open [RPIScratch/Projects/maths_cat](#).

LEARNING OBJECTIVE: In this exercise, you will learn how to put data into a program and get your program to make a decision based on that data.

RESOURCES: The sprites “cat” and “bluedog”; the backgrounds “brick-wall1”, “sydney”, “paris” and “new_york”.

Artificial intelligence is an area of computer science where people try to make computer programs that are smart in some way or another. The idea is to make computers seem like they are thinking like humans. This is actually quite tricky, as you can imagine, so here we are just going to give you a tiny taster of how you can make your programs seem a bit intelligent.

The cat clearly thinks it's clever, so let's give it a chance to show us just how intelligent it is. To do this, we will use some more inputs and outputs, together with something called a “conditional statement”. That sounds very complicated, but it isn't really.

For my example, I have created two sprites: a cat and a dog. The cat is going to ask the dog a number of questions, so we need some variables in which to store the answers.

Create the following variables (we'll tell you what they're for in a minute):

age

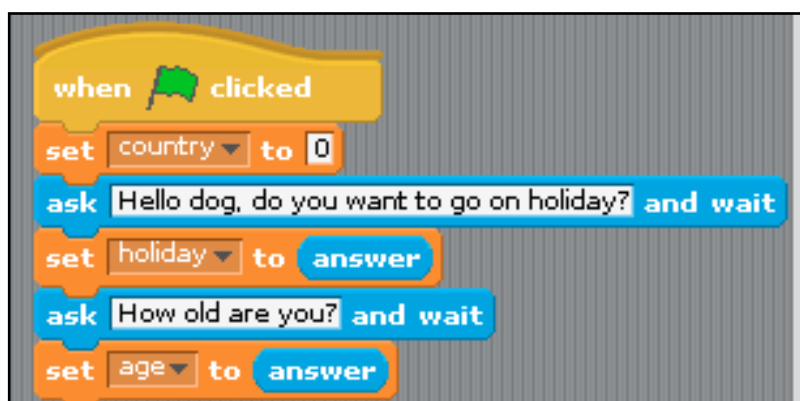
country

holiday

name

Before we begin, you should also import the “bluedog” sprite and the backgrounds “brick-wall1”, “sydney”, “paris” and “new_york”. You will need them for what comes next.

Does the dog want a holiday?



First, we have to work out whether the dog actually wants to go on holiday. Select the cat sprite and build the script you can see in the screenshot above to help the cat find this out.

Each time we use this program the country variable will be set to a letter. But the next time we use the program, we want the country variable to be empty. So we “empty” it by setting it to “0”, ready for the next user.

QUESTION: Study the script. Can you work out what it will do when the program is run?

As the cat asks questions, the dog’s answers are stored in the two variables “holiday” and “age”. The script will use these later.

Now for the intelligent part: the conditional statement. We are going to find out if the dog wants to go on holiday. Time to think logically: the dog will either answer “yes” or “no”.

We need to find a way of letting the cat know the dog’s answer and of prompting the cat to act on that answer.

From Control in the Blocks Palette, select an “if” block.



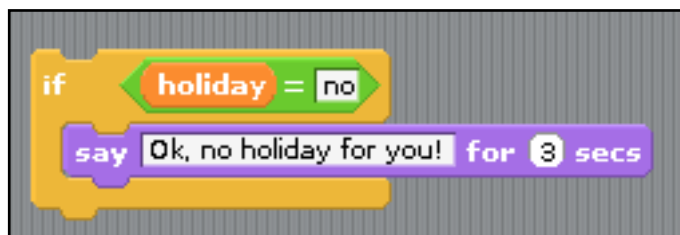
You will also need an operator block from Operators to test the input. Grab the “equals” operator (“[] = []”) from the Blocks Palette.

From Variables in the Blocks Palette, drag and drop the “holiday” variable into the left-hand side of the operator and type “no” into the other side.

Finally, we drag and drop the operator block onto the “if” block.

Now, from Looks place the block “say [Hello] for [2] secs” inside your “if” block – that is, in the “bracket” cut into the side of the block, so that the “if” block surrounds it. Change “Hello” to “OK, no holiday for you!”. Change “2” to “3”.

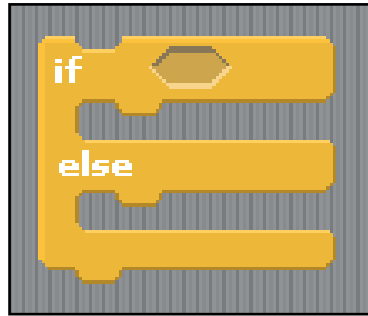
The code to find out if dog wants to go on holiday



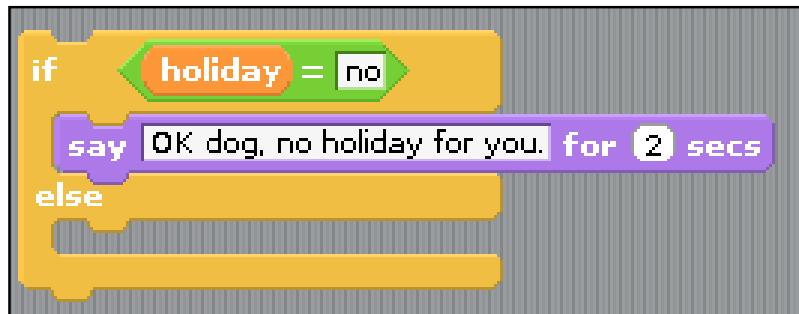
But hang on a second, what if the dog does want to go on holiday? Our “if” block doesn’t allow us to include a possibility for a “yes” answer.

If the dog does want to go on holiday something else must happen. We need to replace our “if” block with an “if/else” block.

The “if/else”
block.



Take the little script that was in your “if” block and put it into the “if” bracket of an “if/else” block. The result should look like the screenshot below.



Notes:

Is the dog old enough to go on holiday?

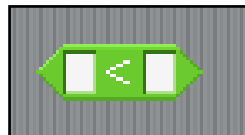
We also need to find out if the dog is old enough to go on holiday by herself. We will need to use a second “if/else” block. Drag it onto the Scripts tab, but don’t attach it to your script just yet – leave it floating by itself.

We’re going to ask the dog how old she is. Depending on her answer, there are two possible outcomes:

1. Her age is less than 10, and she’s not allowed to go on holiday.
2. Her age is greater than 10, and she is allowed to go on holiday.

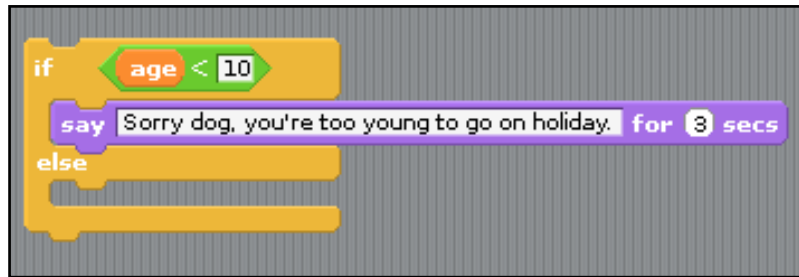
If the dog’s age is less than 10, we want the cat to say, “Sorry dog, you’re too young to go on holiday.”

The “less than”
operator.

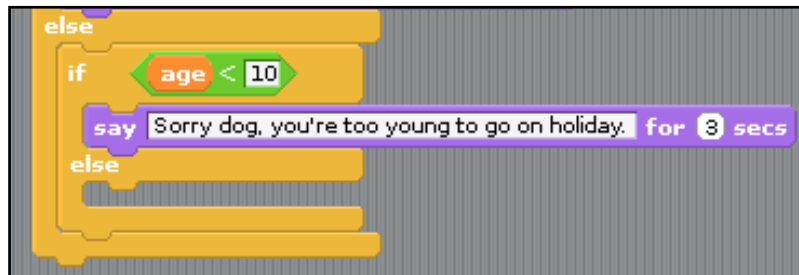


Using your “less than” (<) operator, as well as blocks from Variables and Looks, build the piece of script that you see in the screenshot on the next page.

Build this script inside the “if” bracket of your second “if/else” block.



Before we move on to the next step, take your second “if/else” block and place it inside the “else” bracket of your first “if/else” block: the one you used to find out if the dog wanted to go on holiday.



Your script should now look like the block of code in the screenshot above.

Over to you

QUESTION: What will the cat say if the dog’s age is 9?

QUESTION: What will the cat say if the dog’s age is 12?

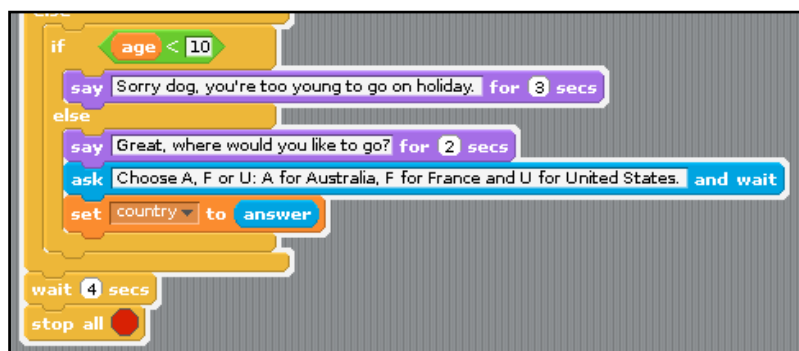
QUESTION: What will the cat say if the dog’s age is 10?

Where does the dog want to go?

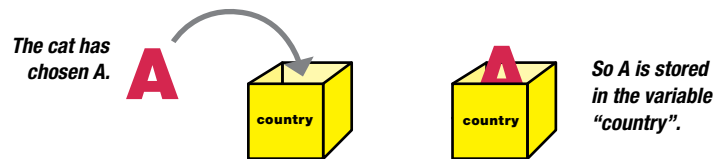
To be fair to the dog, we should let her choose where to go on holiday. We will give the dog three possible holiday destinations: Australia, France and the USA. To keep it simple, we won’t ask the dog to enter the country’s name, just the initial letter.

Using the information you can see in the screenshot below, add blocks to your script to make it ask the dog where she wants to go on holiday. Store the dog’s answer in the variable “country”.

You will need blocks from Looks, Sensing and Variables. Place them in the “else” bracket of your second “if/else” block.



Finally, complete your script with a “wait [] secs” block and “stop all” block from Control. Set the value of the “wait [] secs” to “4” and place both blocks at the very end of your script, outside both “if/else” blocks.



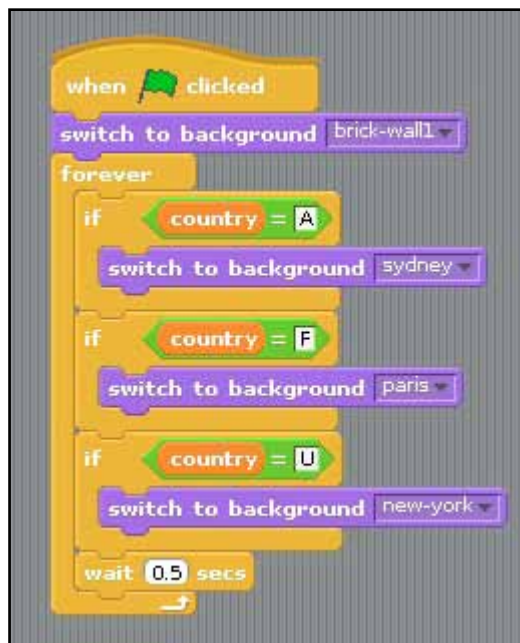
The destination the dog chooses will be stored in the variable “country”. This is important, because this variable will be used by the script for the stage to determine the script’s output.

Notes:

The script for the stage



To complete the program we need to program the backgrounds. If you haven’t imported them from RPiScratch\Resources\Backgrounds, now’s the time to do it.



Select the stage in the Sprites List and build the script you can see in the screenshot above.

Here, we have created three “nested if” conditional statements, which will switch the background according to the dog’s choice. We use the term “**nested**” when one conditional statement is put inside another one.

Each time we use this program, the stage has to be reset to the “brick-wall1” background. The conditional statements need to keep checking until the dog has made her choice, so I have put them in a “forever loop”.



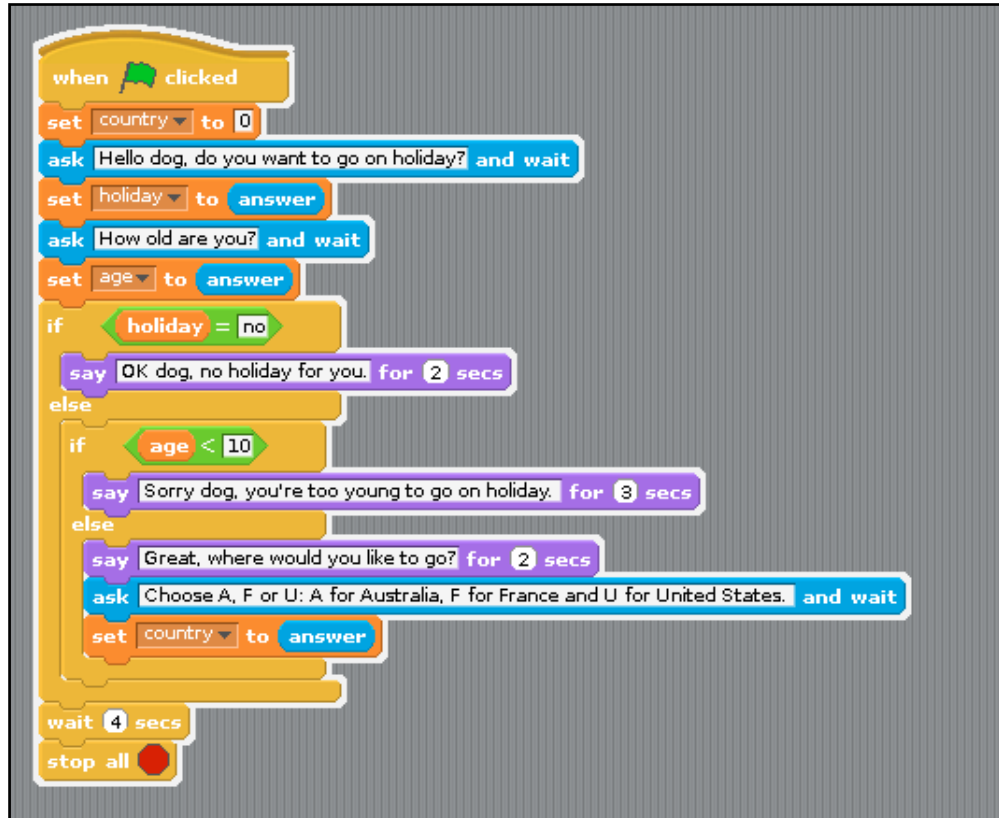
Tip...

Users do not always do what they are supposed to do. If a user provides an unexpected input, this can cause an “error” in the program, which will often cause the program to “crash”. Programs that have errors in them are said to contain “bugs”. An important task for a programmer is testing their program to make sure it is bug-free.

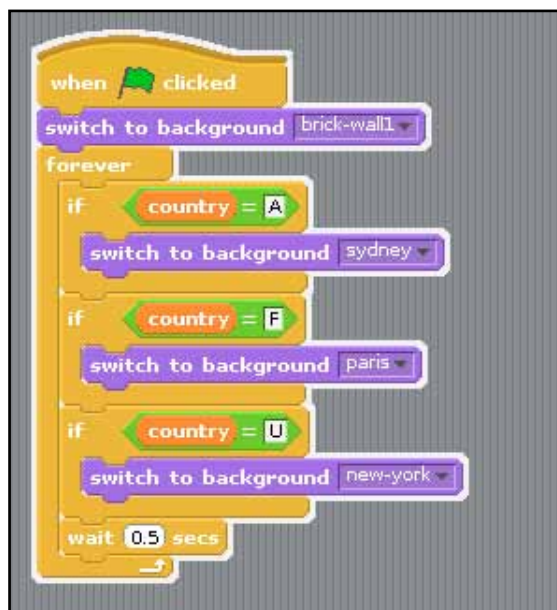
The full script

Phew, that was complicated. But hopefully you managed it all in the end. Just to make things a bit easier, here are the full scripts for both the cat sprite and the stage.

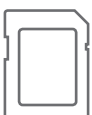
The code for the cat



The code for the stage



Notes:



To see a sample of the code for this lesson, open [RPiScratch/Projects/holiday_dog](#).

Over to you

QUESTION: Study the code shown above for the cat sprite and the stage. Do you understand it?

TASK: Experiment with the code. See if you can add some extra questions.

A final word

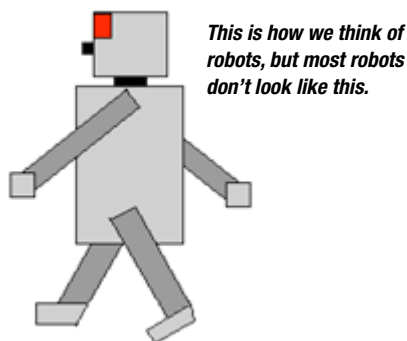
In fact, the cat is not intelligent at all. Computers have no intelligence – it is the programs that run on them that make them appear intelligent. This is why we use the term “artificial intelligence”. The only intelligent things here are the programmer, who programmed the cat, and you, for completing Lesson 1.5.

Lesson 1.6: Control

LEARNING OBJECTIVE: In this exercise, you will learn how to write control programs that respond in different ways depending on inputs to the program.

RESOURCES: The sprite “robot_up” and the background “green_background”.

The robot



You’re probably used to robots from the movies: metal men clanking and talking in metallic voices. This is actually an old-fashioned idea of a robot. It dates from a play written in 1920. I bet even your teacher was young then.

A more modern way of thinking about a robot is as anything that can be controlled by a computer. This is known as “control engineering”. This device could be an aeroplane, a washing machine, a lathe, a welding machine, a level-crossing barrier, a sewing machine, a self-drive car or anything else you can imagine.

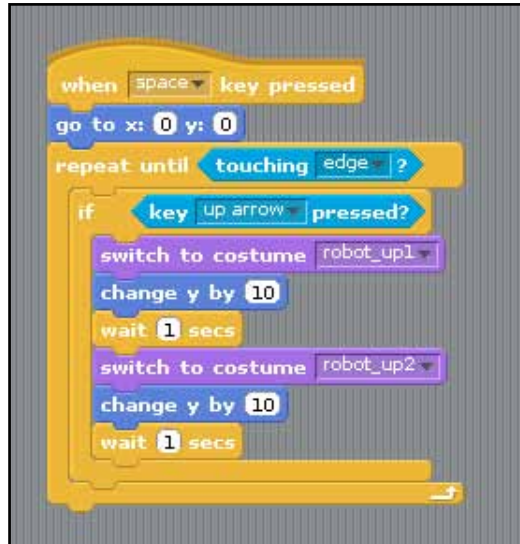
But let’s start with a proper, old-fashioned robot. We don’t have a real robot, or all the wires and circuits we’d need to control one. So, instead, we’ll use a robot that lives inside the Scratch stage. What you learn by doing this is how to make your robot respond to inputs.

We are going to program our robot so that we can control it using the “up arrow” key. To make it easier to see our robot moving, we need to be able to see him from above.



Import the sprite “robot_up” from RPiScratch/Resources/Costumes. While we’re at it, import the background “green_background” from RPiScratch/Resources/Backgrounds.

This script will make our robot move on our command, but only in one direction.



Notes:

Have a look at the screenshot above. We have used the “when [space] key pressed” event from Control to run the program. The robot sprite is moved to the centre of the screen using a “go to x: [0] y: [0]” block from Motion (see “Positioning your sprite”, in Lesson 1.3).

The rest of the code is placed inside a “repeat until” loop (from Control). Inside the repeat loop, we have an “if” (a conditional statement) that checks if the “up arrow” key has been pressed. This has been set to a “key [up arrow] pressed” event (from Sensing).

The “touching []?” block from Sensing is set to “edge”. That means that if the robot touches the edge of the Scratch screen, the event will stop the script. An “event” is something that happens in a program.

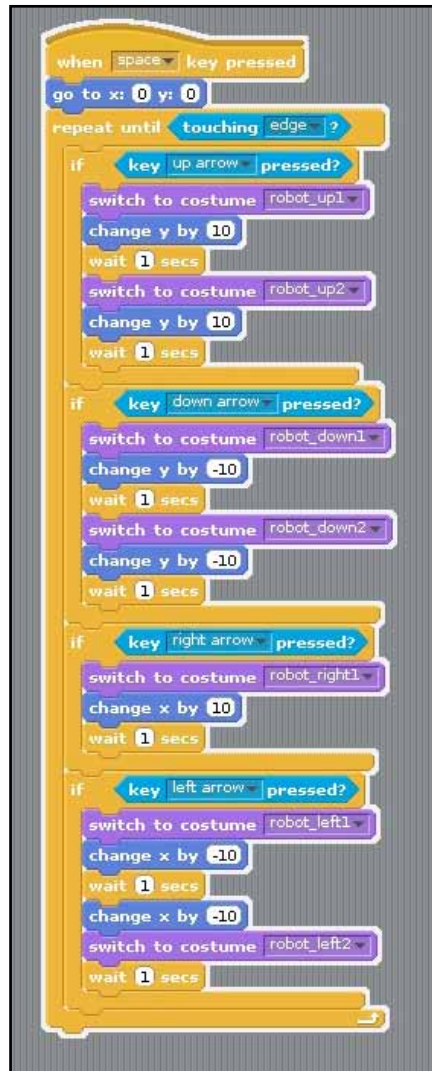
In this script, there are two possible situations:

1. The “up arrow” key has been pressed – in which case, the robot will walk up the screen for 20 steps, switching costumes as it does so.
2. The “up arrow” key has not been pressed – in which case, the robot will do nothing.

That’s not going to be very useful. Let’s see if we can do better.



To see an example of this script, open *RPiScratch/Projects/robot_v1*.

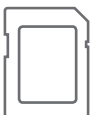


RESOURCES: The sprite “robot_control” and the background “robot_maze”.

We have included the code of a more complete version of the robot program, “Robot_2” (see the screenshot above). In this version, the robot can be moved up, down, left and right.

TASK: Now it’s your turn to do some programming. Program the robot and then use your program to make the robot follow the yellow path on the “robot_maze” background.

TASK: Load the code and then change it to make the robot walk more quickly.



For an example of this script in action, open [RPIScratch/Projects/robot_v2](#).

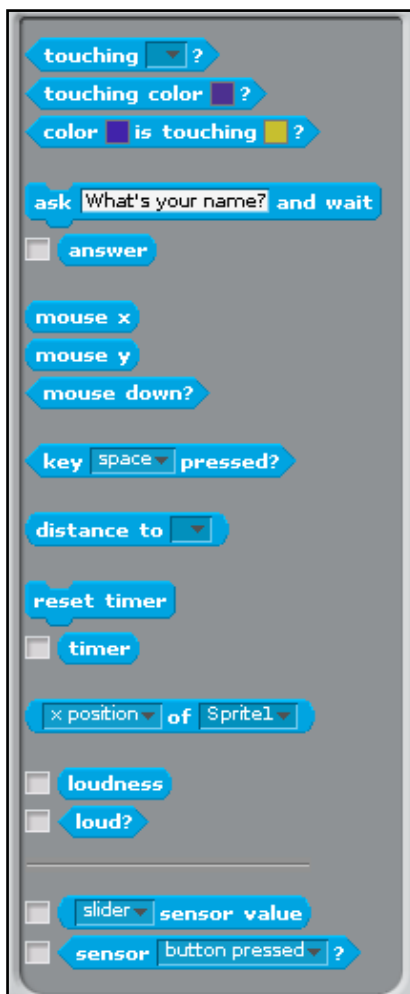
Tip...

This is useful code for controlling characters in games.

RESOURCES: The sprite “yellow_car” and the background “line_background”.

That robot is like something out of a corny science-fiction film. Let’s look at a more modern example. Scientists are experimenting with vehicles that can drive themselves. One way of doing this is by programming the vehicle to follow a line painted on the road surface. These vehicles are known as “line-following vehicles”, or LFVs.

In this exercise, we’ll look in more detail at the blocks in the Sensing section.



The Sensing script blocks allow your program to sense changes in its environment, either in response to user input or events in the program.

Our task is to make a car to follow a line, just like the real engineers designing LFVs for use on the roads. Our main tool is going to be the “color [] is touching []?” block from the Sensing menu.



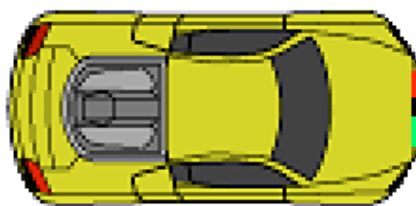
This block allows a program to detect when an area of one colour touches against another colour: for instance, as in the screenshot above, a patch of red touches a patch of black.

Once the program can detect two colours touching, then we can tell it how to respond when this happens. The colour event becomes a trigger, causing the program to do something.

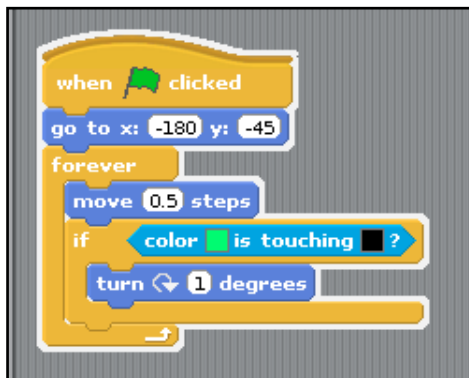
To set the colours in your block, click on one of the little coloured squares. Your mouse-pointer will turn into an “eye-dropper”. Use the eye-dropper to click on the colours you want to use in your block.



But how can we use the ability to detect colour to help make our LFV follow a line on the road? Well, let's start by importing our “line_background” and the sprite “yellow_car”.



Look closely at the car, and you'll see that there's a patch of green on its bumper. We're going to use that green as our first “sensor”. Build the script that you can see in the next screenshot.



Now, click on the green flag and see what happens. Oh dear, everything is fine as long as the line curves to the left but as soon as it curves to the right the car wanders off. What's going on?

The problem is that the car needs two sensors. Currently, if the line bends to the right, the black line will touch the green sensor – the program will detect this and tell the car to turn to the right.

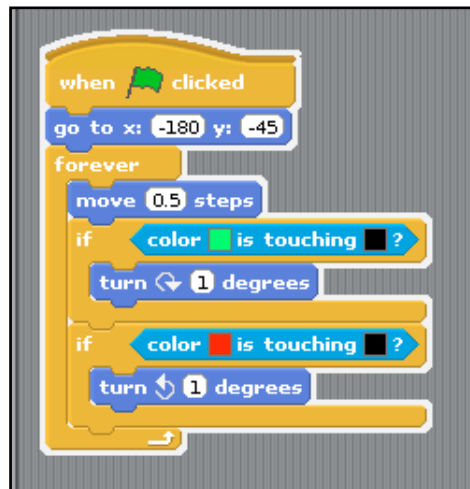
Notes:



If you can't stop your car for long enough to sample a colour, hit the red “stop scripts” button at the top-right of the screen.

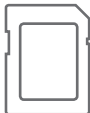
Now, we need to use the red patch on the car's bumper as a red sensor. The program will then be able to detect when the line turns left and tell the car to turn to the left too.

Here's the code you need to finish your LFV program:



As you can see, the program is run with a green flag event. After that:

- A “go to” block is used to position the car on the line.
- We then have used a “forever” loop to keep the car running.
- The car moves 0.5 steps (for each cycle of the loop).
- An “if” block is then used to check if the green spot on the car has touched the black line. If it has, the car will turn to the right by 1 degree.
- Another “if” block is then used to check if the red spot on the car has touched the black line. If it has, the car will turn to the left by 1 degree.



For an example of a completed version of this project, open [RPIScratch/Projects/lfv](#).

Tip...

If your car stops at an odd angle, and you can't straighten it out, you can import the “yellow_car” sprite again, drag and drop your finished script onto the new sprite and then delete the old one.

Tip...

Run the program in full-screen mode – it is more impressive. Click on the right-hand button at the top-right of the Scratch window, just above the green flag and the red circle.

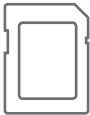
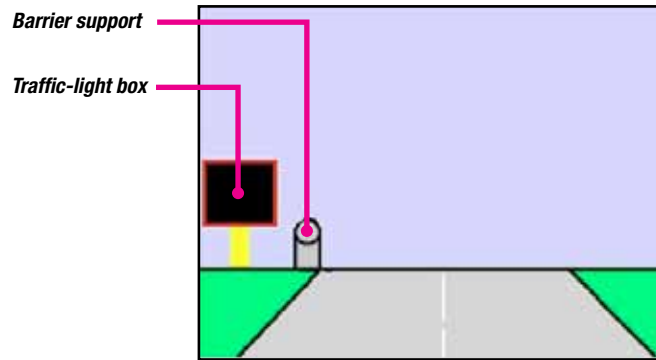
The level crossing

RESOURCES: The sprites “lights” and “barrier”, the background “level_crossing” and the sound “level_crossing_a”.

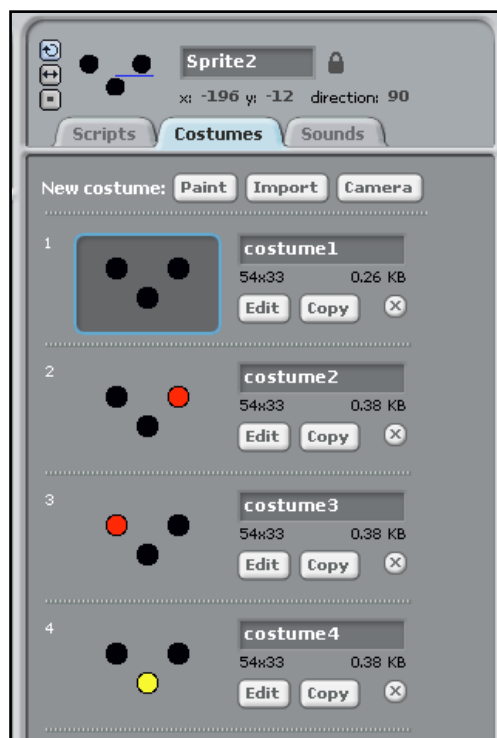
Computer control systems can save lives. At a level crossing we have to make sure that cars and trains never meet. This is done with a sensor on the track that detects the passing train.

A computer then responds to this event by stopping the traffic approaching the level crossing and closing the barrier. This control program can be written in Scratch. But before we start, we need to import the necessary sprites and background.

First, import the background “level_crossing”. As you can see from the picture on the next page, the background shows the road leading up to the crossing, the traffic-light box and the barrier support.



Now we need our sprites. Import “lights” and “barrier” from the RPiScratch/Resources/Costumes folder.



Look at the image above, and you’ll see that the “lights” sprite has four different costumes. By switching costumes quickly, we can make it look as if the lights are flashing.

The last thing we need to do before we start building our script is to import the sound of the level-crossing alarm. To do this, click the Sounds tab, find the sound “level_crossing_a” and then use the import function, just as you did with the background.

Import your sound effect from the Sounds tab. You'll find it right next to the Scripts and Costumes tabs.



Notes:

Now, let's build our script. Before you start, select the "lights" sprite in the Sprites List.

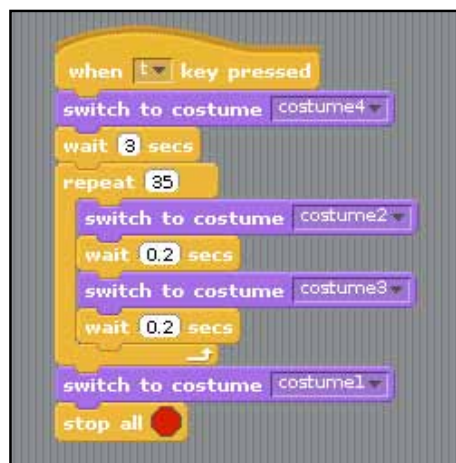
First things first: we want to start with the lights out. So we use a green flag event to switch our lights sprite to "costume1", which shows three black lights.



As the train approaches the level crossing barrier, I have used an "when [] key pressed" block from Control to detect it, setting the value to "t". This event will trigger the lights, alarm and barrier. But first, let's deal with the alarm.

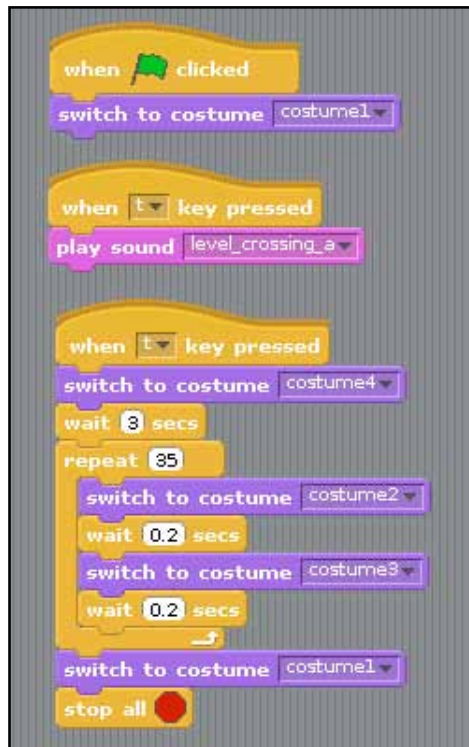


Build the script you can see in the screenshot above. This will cause the alarm to sound when the "t" key is pressed.



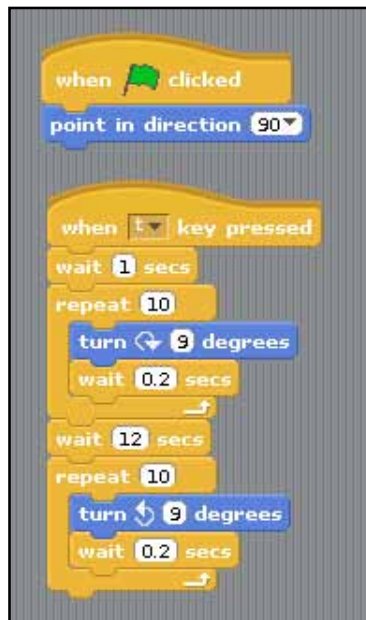
To make the red lights appear to flash we must switch the two red light costumes repeatedly. We use a “repeat” loop for this. Build the script on the previous page. You’ll need to experiment with the number of repeats until you get the timing right.

Notes:



This is the finished program for the lights, with all the blocks in place. Notice that you have three separate scripts, not touching each other.

The barrier

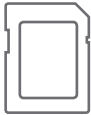


Now for the barrier: drag it until it sits on the barrier support, with its thicker end just resting on the top of the post. Each time the program starts, the barrier must be pointing upwards. Using a “point in direction [90]” block (from Motion) will take care of this.

In the script for the barrier, I have used a “when [] key pressed” block from Control, with its value set to “t”. Think back – the script for the lights also starts when you press the “t” key. In this way, lights and barrier are synchronised: that means they both start at the same time.

To lower the barrier, we will rotate the pole in stages by using a “repeat” block. Experiment with the number of repeats and the angle of turn to move the barrier just the right amount.

After a set time, 13 seconds, we raise the barrier using another repeat block. The train has passed, all the cars are safe. Well done!



For an example of this project in its finished state, open RPiScratch/Projects/level_crossing.

Over to you

TASK: As an extra feature for this program, create a train sprite that moves across the screen when the barrier drops.

Lesson 1.7: Scratch games

LEARNING OBJECTIVE: In this exercise, you will learn some of the techniques used for game programming by playing some Scratch games.

Some of the most fun that you can have with Scratch is through programming games. You can make games about anything you like. We have included two sample games for you to investigate, and there are plenty more games and other examples of programs in the Scratch Projects folder.

The starting point for each new game is the theme or idea, and the aim of the game. When you have an idea for a game, think about the “story” you want to tell and the game’s characters, players, pieces etc. that you will need. These will be the sprites.

You can find great images for sprites and backgrounds by importing them from the Scratch Media folders or by searching on Google Images, saving and importing into Scratch. If you want to get some idea of how to get started and what you can do, have a look at some other people’s games. The best way to learn how to program is to examine other people’s scripts to see how they made cool things happen.

Finally, you will need backgrounds for your stage to create the rooms, levels and scenes in your game, so that your characters have somewhere to move around in.



Tip...

Don’t forget to add sounds and music, as this greatly enhances a game.

Prancing Pony game



RESOURCES: The sprites: “pony” and “girl” and the background “field”.

THE AIM OF THE GAME: A pony moves around a field, making random changes of direction. Using the mouse, the player controls the girl sprite. To score points, you must put the girl on the pony.

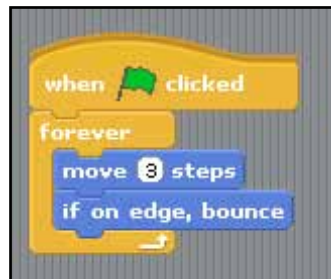
There are four tasks for you to program:

1. Move the pony round the field.
2. Make random changes in direction of the pony.
3. Move the girl with the mouse pointer.
4. Add a point to the score each time the girl is placed on the pony.

So, let's get to it!

Moving the pony

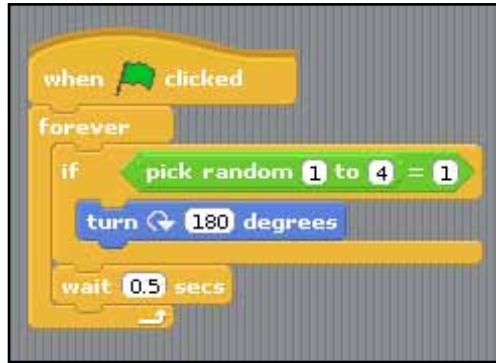
1. Keeping the pony moving



Select the pony sprite and drag a green-flag event to the Scripts tab. We will use the green flag event to run all the scripts in this game at the same time.

We have used a “forever” loop to keep our pony moving (remember your LFV). For each cycle of the loop, the pony moves forward 3 steps. If it hits the edge of the screen it will bounce off in the other direction; this prevents it from getting stuck.

2. Random changes in direction



If the pony never changes direction except when it hits an edge, the game will get a bit dull. So we'll use another "forever" loop to make the pony randomly change direction from time to time.

The "pick random [] to []" Operator block has been set to pick a random number from 1 to 4. If the number picked is 1, the pony will turn 180 degrees.

This is set to happen every 0.5 seconds by the "wait" block.

3. Move the girl with your mouse pointer



Select the girl sprite in the Sprites List. Using yet another "forever" loop, as well as blocks from Motion and Sensing:

- Set the x coordinate of the girl sprite to the x position of the mouse pointer.
- Set the y coordinate of the girl sprite to the y position of the mouse pointer.

4. Score a point each time the girl is placed on the pony.

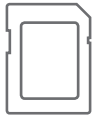


In this script, you can see the following:

- We have created a variable to handle the score and named it "score" (imaginatively!).
- When the game starts the score is set to "0".
- The "forever if" loop checks to see if the red colour of the girl's top is touching a small blue dot on the saddle of the pony.
- Every time the red touches the blue the score is changed to "score + 1".

Hang on a second! Every time the red touches the blue dot, “score” is changed to “score + 1”. If you are good at maths that should set the alarm bells ringing! How can something be itself plus one?

Don’t worry; programmers often do this. Remember that a variable is not a number; it is a container that can store numbers. We can do a sum with the number and put the result back into the same container.



To see a working example of the Prancing Pony game, open [RPIScratch/Projects/prancing_pony](#).



Remember, to stop everything moving so that you can sample the colour of girl’s top with the eye-dropper tool, hit the red “stop scripts” button. If you can’t find the girl sprite, have a look in the bottom-left corner. She may have run off to hide there when you dragged your mouse to the Script tab.

Racing game

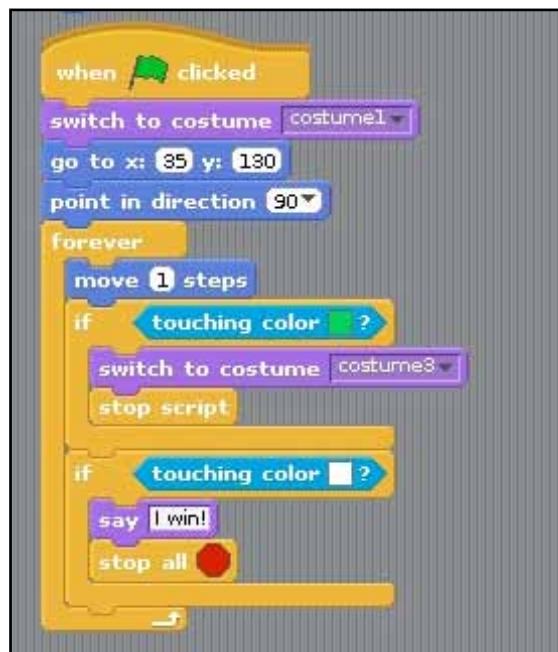
RESOURCES: The sprite “red_car” and the background “track”.

THE AIM OF THE GAME: A two-player game. Each player controls a racing car around a track. If the car goes off the track, it crashes. The first one over the line is the winner.

There are five tasks to program for each car:

1. Line cars up on the starting line.
2. Set the cars in motion.
3. Crash the car if it leaves the track.
4. Declare the first car over the line as the winner.
5. Oh, yes – and the players have to be able to control the cars!

1. Moving the red car



Start off with a green-flag event. When the flag is clicked, the car should:

- switch to “costume1”
- position itself at the coordinates x: 35 y: 130
- point itself to the right (direction 90)

Notes:

To do all this, you'll need, along with the green flag, blocks from Looks and Motion. To move the car, we use our old friend the "forever" loop. The car moves one step each cycle.

If the car leaves the track, it will touch the green grass. This event will cause the first "if" block to crash the car and stop the script for the car.

As the car crosses the finish line, it will touch the white line. This event will cause the second "if" block to say "I win!" and stop all the script running for both cars.

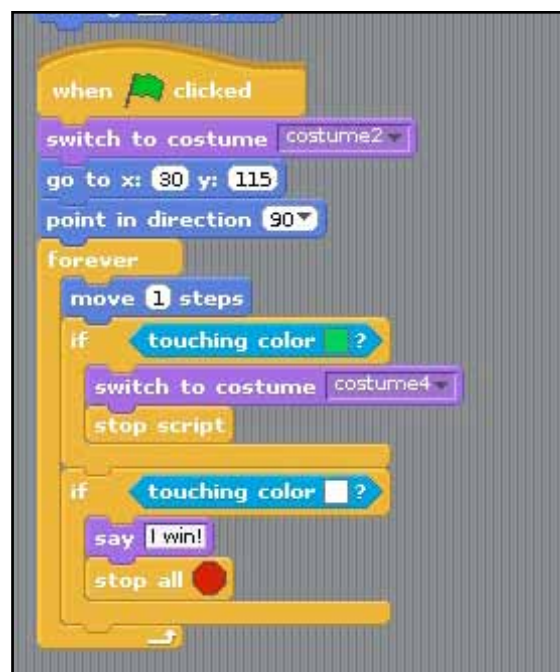
2. Controlling the red car



Ok, so we can get the red car moving. But we need to be able to steer it. To do this we need two "when [] key pressed" (from Control) events are used to control the car.

As you'd expect, the "right arrow" key turns the car to the right and the "left arrow" key turns the car to the left.

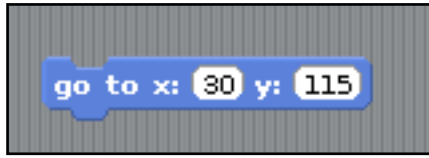
3. Do the same for the purple car



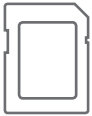
Similar code is used to control the purple car with some slight alterations. The easiest way to do this is to make the red car and then duplicate the sprite. Remember to change the costumes of your new car from red to purple.

The changes you need to make to your code for the purple car are:

The purple car should line up next to the red car at x: 30 y: 115.



You should also use the "x" key to turn the car to the right and the "z" key to turn the car to the left.



To see a finished example of this game, open [RPiScratch/Projects/racing_game_v1](#).

Over to you

TASK: Add the sound of racing cars to enhance the game. You'll need to record or create your own sound file.

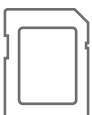
TASK: Try to make these improvements to the racing game:

In the racing game you just created, the car says "I win!" when it hits the line. But the speech bubble is gone so fast, it's hard to read. However, if we change the script so that the speech bubble stays on screen for, say, 2 seconds, then there's a chance that the second car will also hit the finish line in that time and also say "I win!". Then we wouldn't know who had really won.

Try and create a new version of the game, saving it as something like "racing_game_v2", this time causing the losing car to stop in its tracks the moment the winner crosses the finish line. To do this, you'll need to use a variable.

Call this new variable something like "have_winner". The default value of "have_winner" should be "0". Whenever either car wins, by touching the white finish line, it should set the value of "have_winner" to "1". This should be the trigger for both cars to stop moving.

Finally, instead of having the cars crash when they hit the grass, try and make them slow down, so that the player has a chance to get them back onto the track.



To see a finished example of this game, open [RPiScratch/Projects/racing_game_v2](#).

Notes:



Tip...

When you are creating a game, don't be over-ambitious at the beginning. Program your basic idea and make sure that it works. Then develop your game bit by bit, adding new characters, events and levels. It is a good idea to save each new version with a different number, so that you can keep track of the changes that you have made. You may get stuck and want to go back to a previous version of your game.

What next?

Congratulations! You are now a Scratch programmer. Hopefully, this is just the beginning for you, and you will be inspired to create your own programs using the Scratch language. There are many other skills and programming techniques to learn.

Happily, one of the great things about Scratch is the wealth of support and advice on the internet. I have included a few links to some online tutorials for you to explore:

<http://learnscratch.org>

<http://www.scratch.ie>

<http://scratch.redware.com/index.php>

<http://blogs.wsd1.org/etr/?p=395>

<http://scratched.media.mit.edu>

<http://morpheus.cc/ict/scratch/default.htm>

And why not tell your teachers about Scratch? Maybe you and other like-minded kids from your school could even set up a Scratch Club. And when you have projects to do, you could ask your teachers if you can do them using Scratch. The most important thing is to have fun with Scratch.

Stuck and don't know how to get unstuck? Don't suffer in silence. There are some great, friendly forums on the Scratch website where there are lots of other Scratch users to help you.



Build your own blocks

When you become an advanced user of Scratch, you may find that there isn't a script block for a particular job you want to do. Or you might want to improve your program by creating better blocks. Help is at hand with BYOB at <http://byob.berkeley.edu>



Thank you for reading this guide. I hope it helps you to have hours of fun with your Raspberry Pi.

Now it's time to move on to the next chapter, in which you will learn how to program in the Python language as well. Exciting stuff!

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:

Lesson 3.1: Getting to grips with Python

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 >>>):

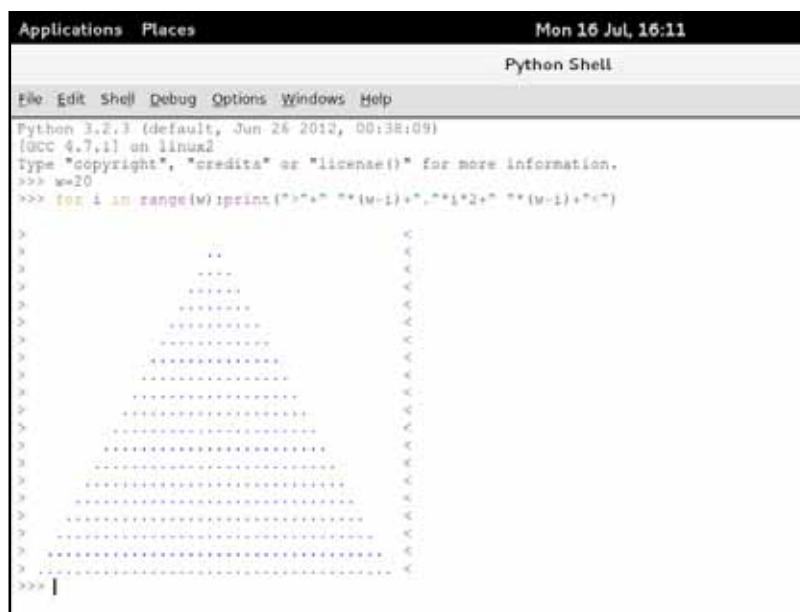
```
>>> w = 20
>>> print (w)
```

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.

A screenshot of the Python Shell window. The title bar says "Python Shell" and the menu bar includes "File", "Edit", "Shell", "Debug", "Options", "Windows", and "Help". The shell text shows the Python version (3.2.3), GCC version (4.7.1), and the execution of the code from the previous block. The output is a pyramid pattern of asterisks and dots, with the number of rows corresponding to the value of 'w' (20). The pattern is symmetrical and grows from 1 row at the top to 20 rows at the bottom.

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

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

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)
```

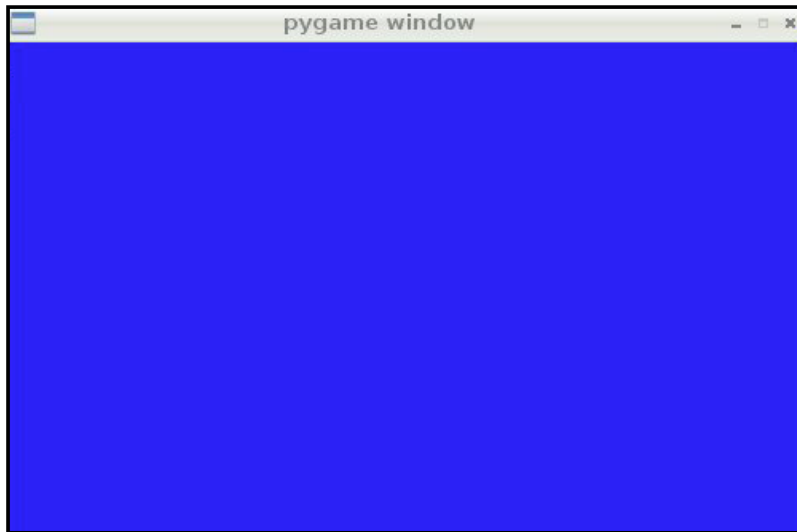
You should now have a display on the screen, which has a surface of 500 x 500 tiny dots, or “**pixels**”. That’s a grand total of 250,000 pixels. Let’s change all 250,000 pixels to a deep blue colour using that “deepblue” tuple we created.

```
>>> 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.



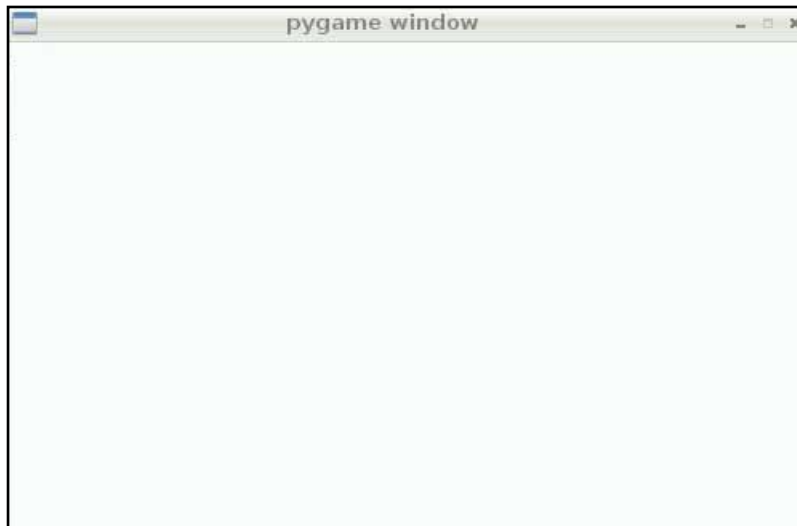
You’ll now have a blue surface in your displayed window. So, that was either fun, okay or really dull. Let’s hope it was at least okay.

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:



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.

Try making up your own colours.

Notes:

```
>>> 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.

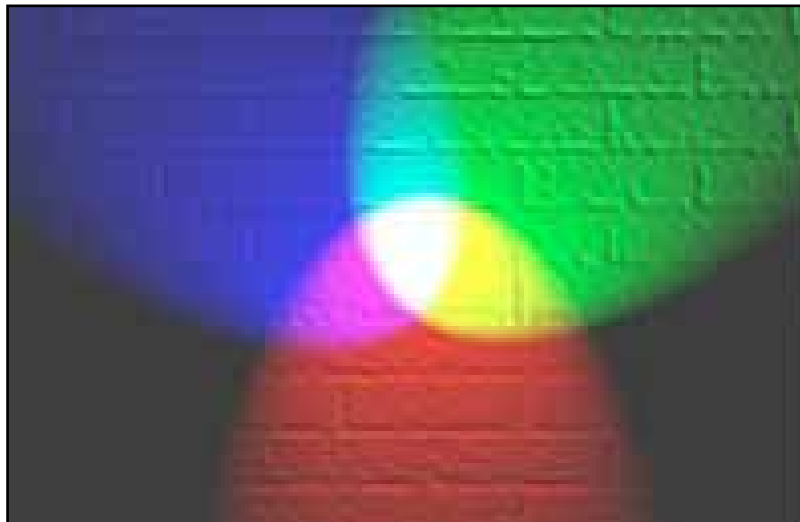
```
>>> 256 * 256 * 256
```

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!*



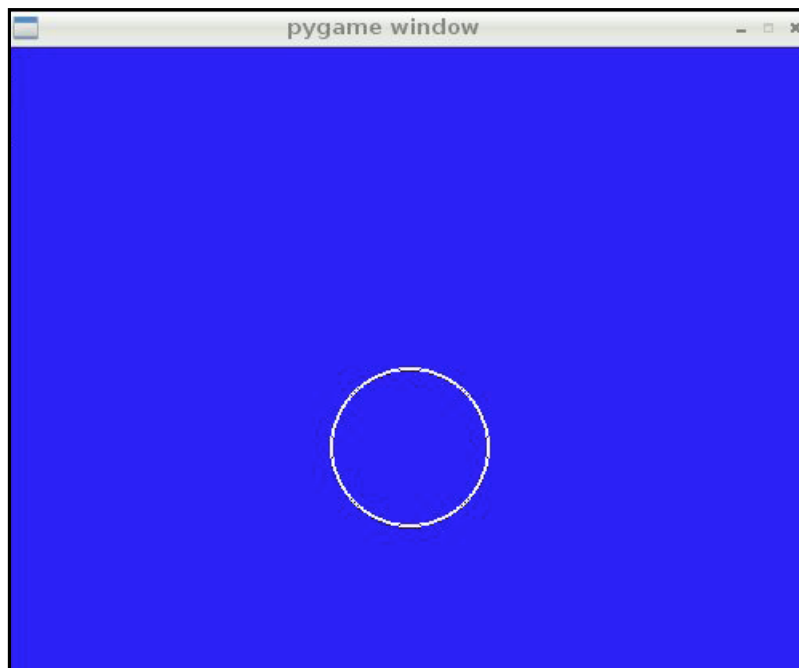
Going round in circles

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()
```



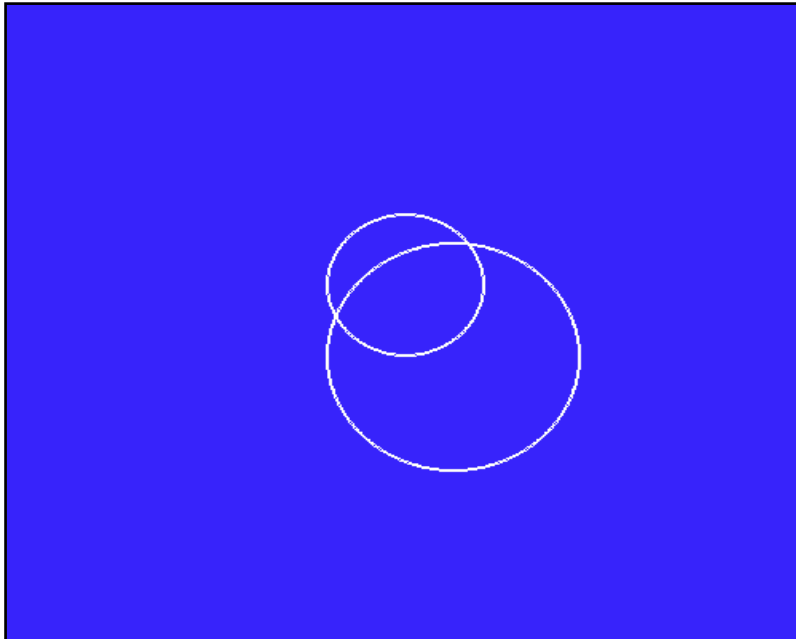
Try changing some of the numbers for drawing the circle and see what happens. Remember to update after each circle you add to the surface.

Notes:



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()
```



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.

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

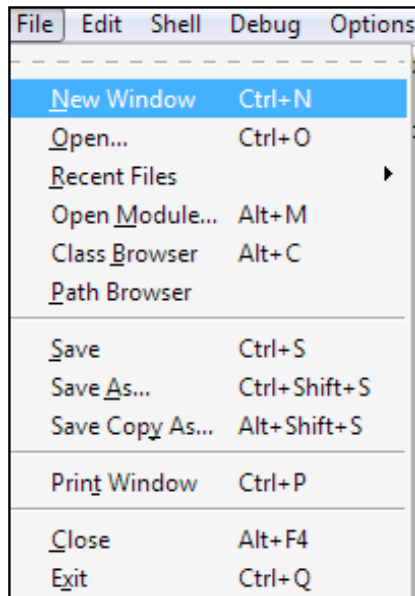
To work out larger numbers, you need to see which of those numbers smaller than it will have no remainder when they divide into it. In the case of 26: 1, 2, 13 and 26 leave no remainder.

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.

Just add Raspberry Pi

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.

Notes:

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.

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*”.

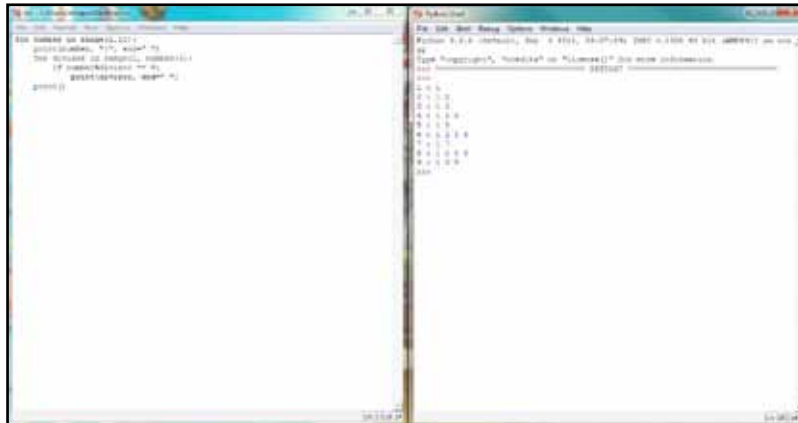
NOTE: It is important to include the “.py” part of this name, so that Python can easily recognise this as a program it can interpret.

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.*



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]
```

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)
... 
```

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:



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.

Lesson 3.2: MasterPy

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)
```

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:



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.

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

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.

```
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.

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.

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)

```

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.

```
# 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)
```

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.

Lesson 3.4: Getting artistic

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.

```
# 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_()
```

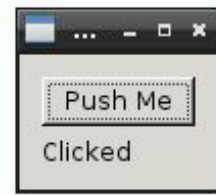
Up to this point, the data being processed has been printed to the screen; but it could have easily been printed on paper or written out to a file. We've been putting text directly onto the console output. If you want to draw onto a window you will need to use a graphical window library. In this section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

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).



```
# 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_()
```

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.

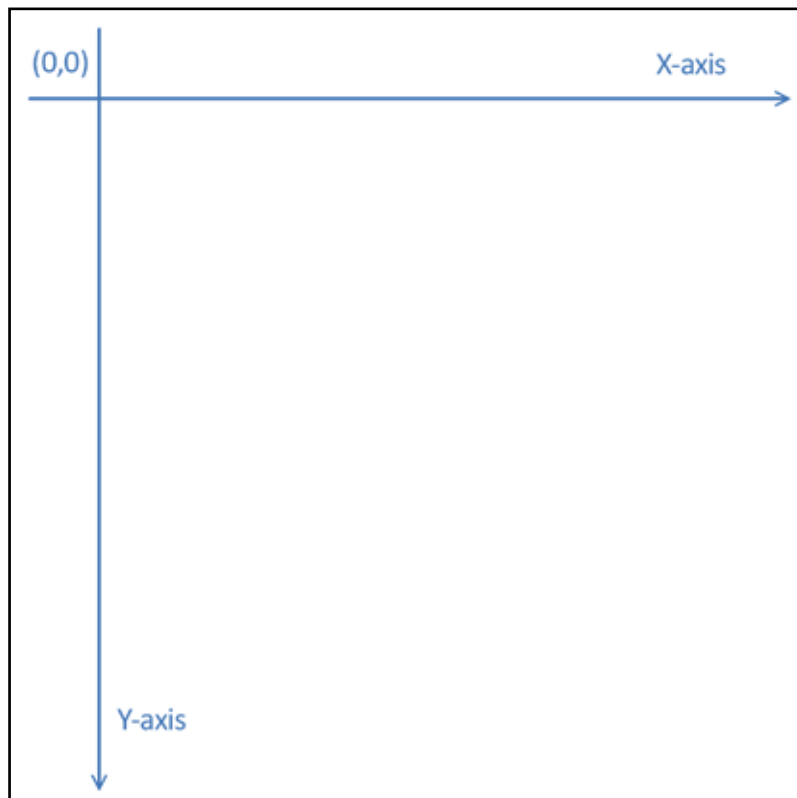
Before going to the main loop, we need to set up a QWidget within the app. This experiment and the last introduced the concept of a “**class**”. Understanding a class is quite complicated but think of it as a block of data that also defines the functions that can manipulate that data. You can create 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.



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

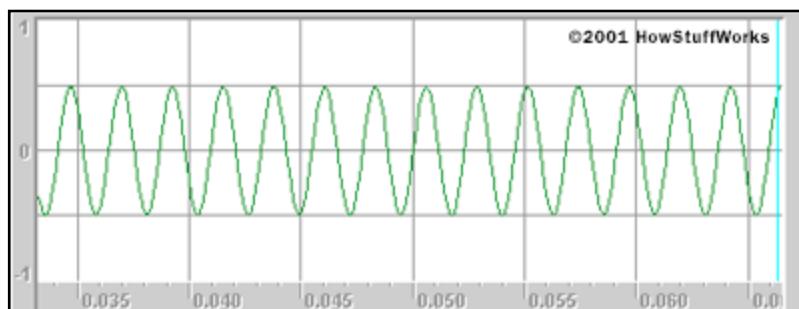
# main program starts here
filename = '/tmp/440hz.wav'
signal = createSignal(frequency=440, duration=4)
createWAVFile(filename, signal)
playWAVFile(filename)
```

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.

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))

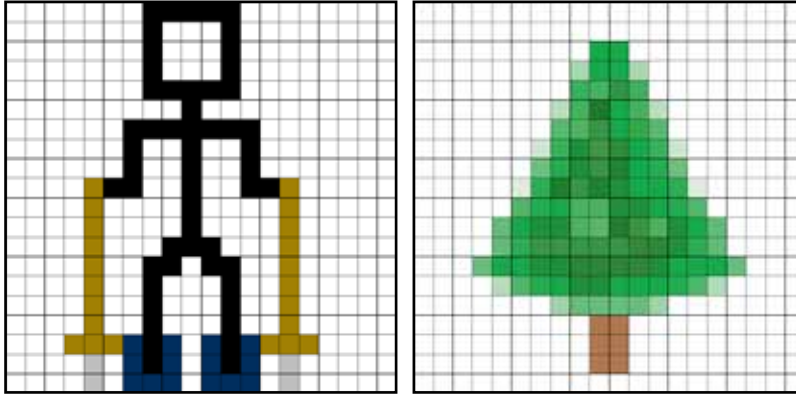
class skiWorld(object):
    running = False
    keydir = 0

    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()

```

```

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.

```
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

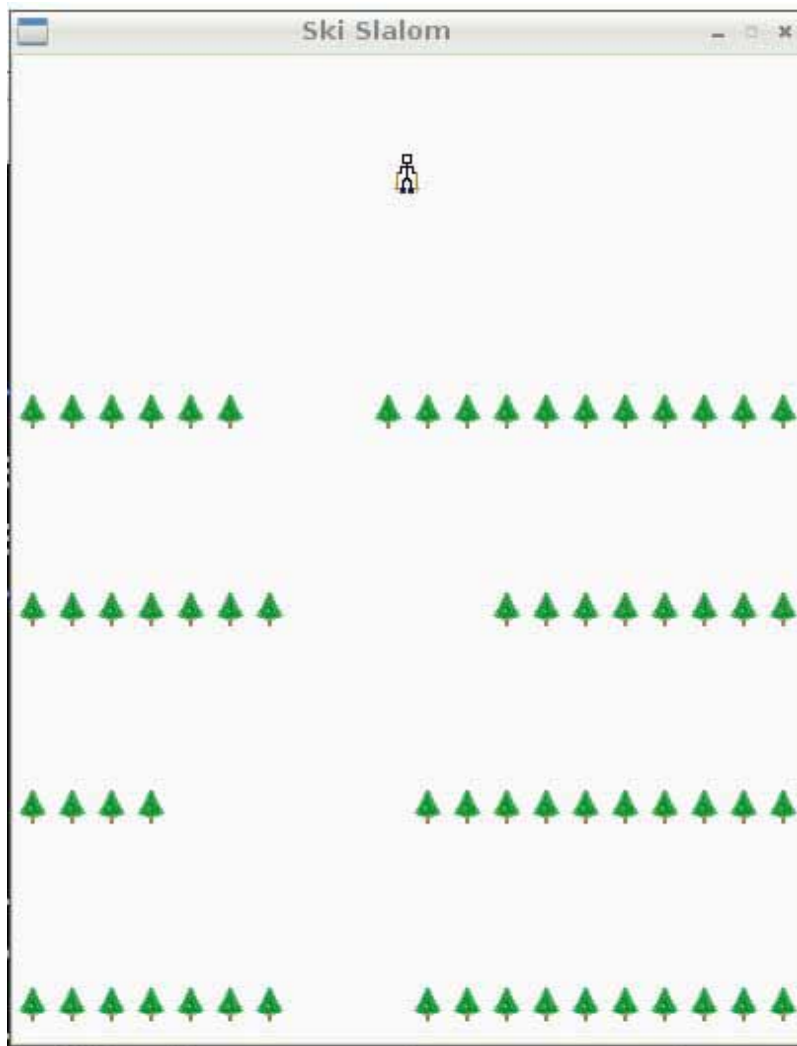
def drawWorld(self, canvas):
    canvas.fill([255, 250, 250]) # make a snowy white background
    world.trees.draw(canvas) # draw the trees
    world.skier.draw(canvas) # draw the player on the screen
```

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:

```
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())
```

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.


```
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.

```

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.

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:

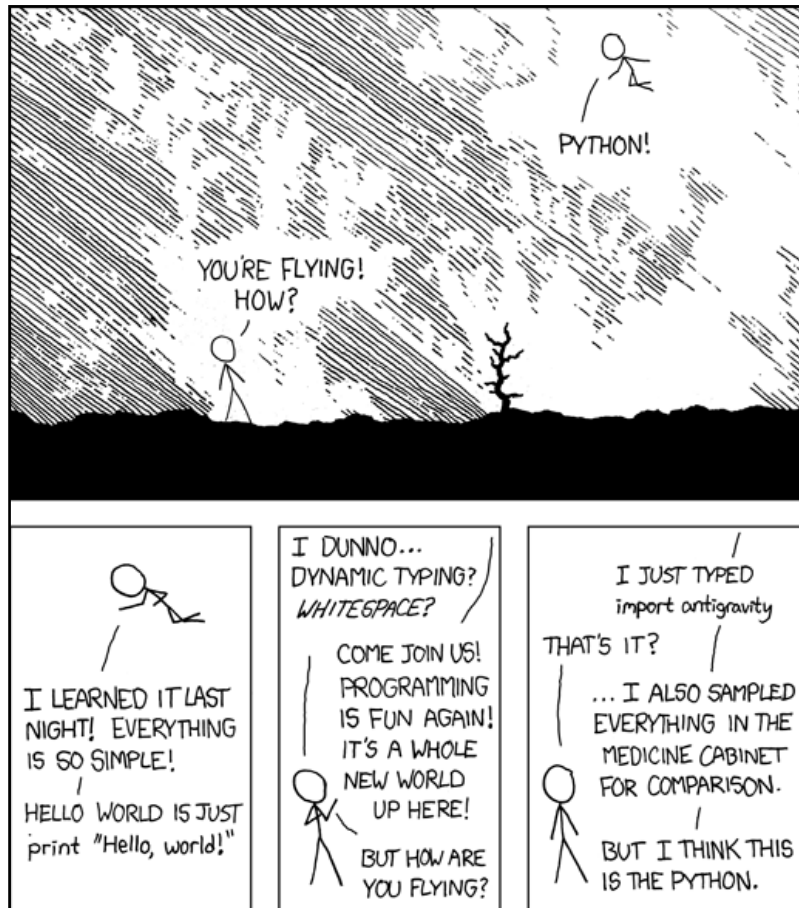
- Beginner's reading
<http://wiki.python.org/moin/BeginnersGuide/NonProgrammers>
- Learning with Python
<http://openbookproject.net/thinkcs/python/english2e/>
- Learn Python The Hard Way
<http://learnpythonthehardway.org/book/>
- Invent with Python
<http://inventwithpython.com/chapters/>
- Python Tutorial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



Human-computer interfacing

Chapter 4

Hear the word “**computer**” and you probably think of a box with a monitor attached, like a desktop PC. But these days, almost every electronic device, from your TV or your parents’ satnav, right up to the auto-pilot in a jumbo jet, is controlled by a computer. Computers are everywhere.

In order to do the job they’re programmed for, computers need to receive information from the real world telling them what to do; these are called “**inputs**”. An input could be a user typing a command or clicking on something with the mouse, or it could be readings from the sensors on an aeroplane wing, telling the auto-pilot the wind speed, air pressure and compass location of the plane. It all depends on the type of computer and the type of program.

To function, computers need inputs – information from an external source. They process this information and produce a result, called an “output”.



Notes:

Fun with ports

Inputs and outputs could also come in the form of a data connection, such as a network link to the internet. Think of the internet, and you probably think of web pages. In fact, the World Wide Web is only one of the many internet applications.

There are lots of other ways to use the internet, such as email, instant messaging, text-based newsgroups or logging on to another computer using the Secure Shell (SSH) network protocol. Each of these different ways of using the internet has a port number associated with it.

By “**port**”, I don’t mean an actual physical connection on your computer, such as a USB or FireWire port. In this context, a port is a software connection. A port can either be open or closed. If it’s open, then that simply means that the program will accept connections over it. If the port is closed, then the program won’t accept connections and the service in question won’t accept any inputs over the network.

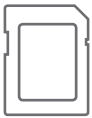
For instance, the HTTP protocol used by the World Wide Web accepts inputs using the TCP port 80. If port 80 is open, a web server will respond to a request from a client, such as your web browser, by displaying the requested web page. If one of that server’s administrators has closed port 80, then you won’t be able to access the website and your browser will display an error message.

Other common ports include TCP port 25, which is used for the SMTP email protocol; ports 20 and 21, which are used for the file transfer protocol (FTP); and port 161, for the SNMP network management protocol. There are thousands of different ports available to use and hence thousands of different ways to communicate on the internet!

In this chapter, we are going to use your Raspberry Pi to interface with a range of internet applications, including Twitter and email. We'll communicate with each program via the appropriate port, providing input that causes that program to run a specific function. And we're going to do it all using the Python programming language. You won't touch a browser once.

All the examples in this section are designed to be as simple as possible. Each exercise demonstrates just one type of communication. Think of these examples as ingredients in a recipe that, when mixed together, produce something much better than the individual parts. The extent of what you make is only limited by your imagination! You have your Raspberry Pi – now get cooking!

What tools will I need?



To complete the exercises in this chapter you will need the “**IDLE**” program that you used in the last chapter. Some of the exercises also require you to install extra Python modules. I'll tell you when this is the case. Wherever possible, I have made these extras available on the SD card that came with your Raspberry Pi, so look out for the SD card logo!

These resources are also available on Google Drive, here: <http://goo.gl/vK3VP>

Lesson 4.1: Twitter



To complete this exercise you will need to install the Python Twitter Tools. You can either download these from <http://pypi.python.org/pypi/twitter/> or install them from the SD card that came with your Raspberry Pi.

The following Python script will tweet the message “Hello, World!” from your Raspberry Pi. It then displays a list of all your personal tweets. No web browser in sight! This example shows how simple this type of application can be, using Python.

```
import os
from twitter import *

# go to https://dev.twitter.com/apps/new to create your own
# CONSUMER_KEY and CONSUMER_SECRET
# Note that you need to set the access level to read and write
# for this script to work (Found in the settings tab once you
# have created a new application)
CONSUMER_KEY = '9HvofZMpvHo0KGZyg9ckg'
CONSUMER_SECRET = 'S75MwXN2H0h2qIYszc51WtHTbpbouhDkr6CCTBPzA'
# get full pathname of .twitterdemo_oauth file in the
# home directory of the current user
oauth_filename = os.path.join(os.path.expanduser('~'),
                              '.twitterdemo_oauth')

# get twitter account login info
if not os.path.exists(oauth_filename):
    oauth_dance('Raspberry Pi Twitter Demo', CONSUMER_KEY,
               CONSUMER_SECRET, oauth_filename)
    (oauth_token, oauth_token_secret) = read_token_file(oauth_filename)

# log in to Twitter
auth = OAuth(oauth_token, oauth_token_secret, CONSUMER_KEY,
             CONSUMER_SECRET)
twitter = Twitter(auth=auth)

# Tweet a new status update
twitter.statuses.update(status="Hello, World!")

# Display all my tweets
for tweet in twitter.statuses.user_timeline():
    print('Created at', tweet['created_at'])
    print(tweet['text'])
    print('-'*80)
```

Notes:



Tip...

As in the previous chapter, we sometimes have more code to write than fits on one line. When you see a line of code that goes onto a new line, don't press Return to start a new line, just let it word wrap and Python will work it out for you.

Over to you

Try to think of other ways in which you could use this mechanism. For instance, take an input from somewhere, process it and tweet the result. You could detect the weather outside and tweet the result. You could even make a tweet from your mobile phone and have your Raspberry Pi read and react to it (by playing some music, for example)!

Lesson 4.2: Email application

Email is transferred across the internet using something called SMTP – Simple Mail Transfer Protocol. SMTP uses port 25. Don't worry; you don't need to understand how SMTP works to be able to send an email. There is a Python module that does all the hard work for you.

Use the Python script below to send a single email.

email-example.py:

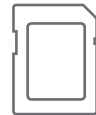
```
import smtplib
from email.mime.text import MIMEText

# create the email
message = """This is a test of using Python on Raspberry Pi
to send an email. Have fun!"""
msg = MIMEText(message)
msg['Subject'] = 'RPi Python test'
msg['From'] = 'My RPi <my_rpi@example.com>'
msg['To'] = 'you@yourdomain.com'

# send the email
s = smtplib.SMTP('smtpserver')
s.login('username', 'password')
s.sendmail(msg['From'], msg['To'], msg.as_string())
s.quit()
```

You can modify the program to send email attachments as well as simple text. The example on the next page sends an image (such as a JPG file) as an attachment to the message.

Notes:



email-attachment.py:

Notes:



```
import smtplib
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
from email.mime.image import MIMEImage

# create multipart email
msg = MIMEMultipart()
msg['Subject'] = 'Lily'
msg['From'] = 'My RPi <my_rpi@example.com>'
msg['To'] = 'you@yourdomain.com'
msg.preamble = 'This is a multi-part message in MIME format.'

# attach email text
message = """This is a test of using Python on Raspberry Pi
to send an email. This email also includes a picture as an
email attachment. Have fun!"""
msg.attach(MIMEText(message))

# attach a JPG file
filename = 'picture.jpg'
with open(filename, 'rb') as f:
    img = MIMEImage(f.read())
img.add_header('Content-Disposition', 'attachment', filename=filename)
msg.attach(img)

# send email
s = smtplib.SMTP('smtpserver')
s.login('username', 'password')
s.send_message(msg)
s.quit()
```

Over to you

Again, try and think of other ways you could use this script. For instance, you could generate the content of the email (the output) based on an input. For example, checking the room temperature and emailing a warning if it is getting too hot. This is regularly used in server rooms, for instance, to detect when the air conditioning has failed.

NOTE: You will have to change the text highlighted in yellow to your own email address and email server details for this program to work.

There is more information about the email modules in Python available at <http://docs.python.org/py3k/library/email.html>

Lesson 4.3: Remote Procedure Call

Notes:

A **Remote Procedure Call (RPC)** is bit of code in one program that causes another program to do something, for instance run a subroutine (a bit of a program that performs a specific task). The two programs don't necessarily have to be on the same computer, although often they are.

When using RPC, calling a subroutine looks the same in your code as if it was being executed as part of your program. The communication between programs remains largely hidden and the programmer does not have to worry about the details of how it is done.

The language the programs are written in and type of computers do not have to be the same either – they can be on completely different hardware, operating systems and programming languages. For example, a mobile phone app could be programmed to call a Python program running on a Raspberry Pi to perform a task and return a result.

The type of Remote Procedure Call we are using for this example is called **XMLRPC**. Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding data in a format that is both human-readable and machine-readable.

There are two programs here – a “**server**” and a “**client**”. The server sits waiting for one or more clients to ask it to do something.

rpc_server.py:

```
from xmlrpc.server import SimpleXMLRPCServer

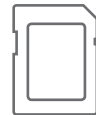
def hello(name='World'):
    """Return a string saying hello to the name given"""
    return 'Hello, %s!' % name

def add(x, y):
    """Add two variables together and return the result"""
    return x + y

# set up the server
server = SimpleXMLRPCServer(("localhost", 8000))

# register our functions
server.register_function(hello)
server.register_function(add)

# Run the server's main loop
server.serve_forever()
```



rpc_client.py:

```
import xmlrpc.client

s = xmlrpc.client.ServerProxy('http://localhost:8000')

# call the 'hello' function with no parameters
print(s.hello())

# call the 'hello' function with a name
print(s.hello('XMLRPC'))

# add two numbers together
print('2+3=%s' % s.add(2,3))
```

To run this example, you need two separate IDLE windows – one for the server and one for the client. You need to run the server first so that it is waiting for requests from the client. To stop the server, press Ctrl-C.

How does it work? The server program sits and waits for requests from clients. The client program, when run, calls the “hello” and “add” subroutines. The calls are passed from the client to the server program, where the functions are actually run. The result is then returned to the client program, which displays the result on screen.

Where could this XMLRPC mechanism be used? Well, imagine that you have several Raspberry Pis, each one connected to a screen located in a different room. You also have a central computer running a control application with a Graphical User Interface (GUI) designed to talk to every room over the network.

This central application could monitor and control every room by communicating with individual Raspberry Pis – telling them what to display on the screen, control devices via GPIO, and return data recorded from the room (such as light levels or temperature). You could even take pictures with a camera! Each Raspberry Pi would run as an XMLRPC server; the central control program would act as the client.

Over to you

You can read more about Python’s XMLRPC modules here:

<http://docs.python.org/py3k/library/xmlrpc.server.html>

<http://docs.python.org/py3k/library/xmlrpc.client.html>

There are, however, XMLRPC libraries available in lots of other programming languages.

Notes:



A **web application** is a program that communicates on the World Wide Web. Like any other program, it has inputs, processes them and produces some output. Not every input and output from a web application has to be through a web browser, though!

The World Wide Web uses something called **HTTP** (Hyper Text Transfer Protocol). This is what web servers and web browsers use to communicate with each other at a low level. The port number used for HTTP is 80. The detail of how HTTP works does not need to concern us at this stage – all the work is done for us by web browsers and libraries in Python.

A **URL** (Uniform Resource Locator), otherwise known as a web address, is what you would type into the address bar of a web browser. It can be made up of a few parts:

protocol :// hostname : port / address

Let's look at those parts in order:

Protocol

For a website, this is normally "HTTP:" but doesn't have to be. Secure websites use the encrypted "HTTPS:" protocol, and most browsers can also work with the file transfer protocol ("FTP:").

Hostname

This is the name of the computer, the web server, on which the website resides. In these examples, we use the hostname "localhost", which means the computer on which a program resides.


Port

This is a number between 1 and 65,535, specifying the TCP port to be used for the communication. If not specified, the default port, 80, is used.

Address

This is the address of the specific webpage that you want to view. For instance, in the URL **www.raspberrypi.org/faqs** the suffix "faqs" is the address.

HTML (Hypertext Markup Language) is a language used to describe the layout of a web page. In both of these examples here, you will notice that the web page created is fairly basic. Imagine how you can improve this just by modifying the HTML!

 **Tip...**

If you want to find out the **IP address** of your Raspberry Pi, open the Terminal, type the command "ifconfig" and press Return. This command does the same job as "ipconfig" on a MS Windows computer.

Inputs

The main input to a web application is in the form of a “**page request**” from a web browser. As part of this request, variables are passed from the web browser to the web server (in this case, your program). These variables might include things such as the specific page you want to see, login details, details about how you would like the site to look and so on.

Normally, these variables exist only for the lifetime of one request. Once you close your browser, they’re gone, and if you visit the site again, you’ll have to specify the details of your request to the server all over again. If you want variables that persist between requests, you can use a special type of variable called a “**cookie**”. Cookies are stored by the web browser and remembered between requests.

Your application’s inputs can be more than just page requests from a web browser though – a database is often used to store data, for example. You could even use the temperature of the room from a thermometer connected to your Raspberry Pi!

Outputs

Outputs from a web application are called a “**response**”. The output is normally in the form of the HTML of a web page, although it could be any data type, such as a JPG file (a picture). The content of this output is generated by the web application. You could even make your application produce a completely different type of output as well as a web page – for example, changing the message on a large electronic sign connected to your Raspberry Pi.

Processing

On the following pages are two simple Python programs that run as web servers. This code will also run without modification on a commercial web server running Apache web server software and the `mod_wsgi` Python module.



To complete this exercise, you will need to download and install the “WebOb 1.2” Python module. You’ll find this either on the Raspberry Pi SD card or at the web address <http://pypi.python.org/pypi/WebOb/>

```
from webob import Request, Response

class WebApp(object):
    def __call__(self, environ, start_response):
        # capture the request (input)
        req = Request(environ)

        # get the name variable,
        # default value of 'World' if it is not set
        name = req.params.get('name', 'World')

        # generate the HTML for the response (output)
        html = """
<p>Hello %s!</p>
<form method="post">
Enter your name: <input type="text" name="name">
<input type="submit" value="Submit Form">
</form>
"""

        # create and return the response
        resp = Response(html % name)
        return resp(environ, start_response)

application = WebApp()

# main program - the web server
if __name__ == '__main__':
    from wsgiref.simple_server import make_server
    port = 8080

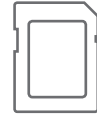
    httpd = make_server('', port, application)
    print('Serving HTTP on port %s...' % port)
    httpd.serve_forever()
```

Run this application in IDLE, then launch a web browser. Go to the URL “http://localhost:8080”. If you’re working from another computer available on the same network, change “localhost” to the IP address of your Raspberry Pi. Remember, you can find your IP address by typing “ifconfig” into the Terminal. You can stop the server by pressing Ctrl-C.

Note that most web pages are normally served on port 80. In order to run on a port below 1024, your program must be run with “superuser” (root) privileges. We have used port 8080 with this program – this is commonly used by web applications that are run as a normal user.

How does it work? The “name” variable is passed through to your program in the form of a “POST” request. This is generated using a form in HTML. The first time your page is viewed, the “name” variable is not set, so a default value of “World” is used.

We have just seen an example using a simple form. You may have noticed a drawback – the “name” variable from that form only lasts for the lifetime of one request, then it is lost. If we want a variable to remain between requests, we have to use another type of variable called a “**cookie**”. The program below is a modified version of the program above that demonstrates how to use cookies.

cookies.py:

```
from webob import Request, Response

class WebApp(object):
    def __call__(self, environ, start_response):
        # capture the request (input)
        req = Request(environ)

        # get the cookiechange variable,
        # default value is an empty string
        cookiechange = req.params.get('cookiechange', '')

        if len(cookiechange.strip()) > 0:
            # change the value of cookie if cookiechange box
            # has been completed
            cookie = cookiechange
        else:
            # get the cookie, default value is an empty string
            cookie = req.cookies.get('cookie', '')

        # generate the HTML for the response (output)
        html = """
<p>The cookie is set to '%s'</p>
<form method="post">
Change cookie value: <input type="text" name="cookiechange">
<input type="submit">
</form>
"""

        # create the response variable
        resp = Response(html % cookie)

        # store the cookie as part of the response,
        # max age 30 days (=60*60*24*30 seconds)
        resp.set_cookie('cookie', cookie, max_age=60*60*24*30)

        return resp(environ, start_response)

application = WebApp()

# main program - the web server
if __name__ == '__main__':
    from wsgiref.simple_server import make_server
    port = 8080

    httpd = make_server('', port, application)
    print('Serving HTTP on port %s...' % port)
    httpd.serve_forever()
```


When using this program, you will notice that the value of the cookie is remembered even if you close the web browser.

How does it work? First, we display the current value of the cookie. If the “cookiechange” variable is set, the value of the cookie is changed and this is sent back to the web browser to store. The cookie has a maximum age though – we have used 30 days in this example. The cookie may be lost even sooner if it is deleted by the user in the web browser. The web browser may even be set up to ignore cookies completely and never store them, although this is not usually the case.

Further reading

More information about the “WebOb” library can be found at
<http://docs.webob.org/en/latest/>

There are also several Python web frameworks available that do a lot of the work for you. A framework is recommended if you are thinking about writing an entire website. A popular example is the Django framework; see
<http://www.djangoproject.com/>

We have only given a very small taster of what is possible here – there is a huge amount that can be learned about writing web applications.

Lesson 4.5: General Purpose Input/Output (GPIO)

GPIO is short for “**General Purpose Input/Output**”. Your keyboard, mouse and monitor are examples of input and output devices on a computer, but they are for specialised and well-defined tasks. The “General” part of GPIO indicates that you can design your own device and connect it up to the Raspberry Pi.

This section explains the technicalities of how inputs and outputs are connected and processed by computers. Following that, there is a technical reference of the GPIO capabilities of a Raspberry Pi. I'll finish off with some simple electronic circuits and software that you can build and use on your Raspberry Pi. In this guide, we are going to be concentrating on the digital GPIO interface, which is the simplest to use and understand as a beginner.

So, let's begin by looking at how computers receive and use information.

Sensors and output devices

To produce useful results, computers need data to work with. Data comes from inputs, and inputs arrive via sensors, of one kind or another.



Inputs are pieces of information sent into a computer, much like a human can feel, smell, taste, hear and see things. Outputs are the pieces of information that are produced and sent out by a computer, much as you can speak or gesture. As computers are electrical, inputs and outputs must be converted to and from an electrical form so that the computer can work with them. A sensor is a piece of electrical hardware that detects and converts something in the real world (such as the speed of a wheel) into an electrical signal. An output device is something that converts an electrical signal to another form (such as a light, a buzzer or a motor).

Digital and analogue input/output

A digital signal is one that can exist in one of only two states, such as a light switch that is either on or off. While one digital channel (digit) can be either on (“1”) or off (“0”), we can combine several digits to make a number. This is the binary system covered in the Python chapter – each binary digit represents one digital channel.

With two digital bits we can create:

Binary	Decimal
0	0
1	1
10	2
11	3

Notes:

Notes:

Notes:

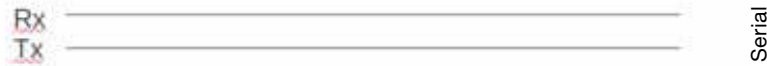
Notes:

The diagram illustrates the process of analog-to-digital and digital-to-analog conversion. On the left, an analog signal (represented by a single line) enters an ADC (Analog-to-Digital Converter), which outputs a digital signal (represented by multiple lines). On the right, a digital signal (represented by multiple lines) enters a DAC (Digital-to-Analog Converter), which outputs an analog signal (represented by a single line).

Notes:

Serial and parallel data

In serial data signals, there are two information streams: transmit (Tx) and receive (Rx).



In a parallel signal, many bits of data can be sent or received simultaneously.



Digital data flows in and out of a computer in one of two ways – in serial or in parallel. On a parallel interface, there is one data channel for each bit of data that is input. Think of it as having lots of lanes on a motorway. On a serial interface, there is just one data channel for each direction (Transmit and Receive), which every single data bit travels along, one after the other. Think of it like a single road. Parallel interfaces are simpler to design and use, but require more hardware and data cable to construct (much like the motorway taking up more space than the road).

Interrupts, polling and multitasking

Now we know the basics of how information (data) flows in and out of a computer, we have to know how to handle the data that is coming in when it arrives. Consider this situation: You are working on your computer but, at the same time, you are also expecting one or more phone calls. Think of the phone call as a source of input data. There are three ways to be able to deal with both tasks at the same time:

- 1.** Work on the computer until you are interrupted by the phone ringing. Answer the phone then continue working on your computer.
- 2.** Work on the computer for a minute. Pick up the phone and check to see if there is someone on the other end. Put the phone down then do another minute of work on the computer. Check the phone again. Repeat this all day!
- 3.** Work on the computer yourself and get another person to take phone calls for you.

The first method is an example of using “**interrupts**”. When on the phone, you could write down a note for yourself so that you can prioritise any work that occurs as a result of the phone call in a timely manner. Using interrupts is the most efficient use of time with infrequent events. There is, however, a disadvantage to using interrupts: it requires low-level hardware or software (ringtone on the phone) to be available.

The second method is an example of “**polling**”. It tends to waste a lot of your time by frequently checking the phone when there is nobody there. It also takes time to swap from one task to the other. This is what you would have to do if the phone had no ringtone. Polling is the method that is used when events occur frequently and must be handled in a timely manner.

The third method is an example of “**multitasking**”. The first person can concentrate on their work on the computer. The second person could be checking (polling) the phone all the time and pass messages to you as and when needed. The problem is that there is not always a second person around. Don’t worry – operating systems such as Linux on the Raspberry Pi do make it possible for you to do multitasking, even though there is only one processor.

Embedded applications

An embedded application is where a computer is built into another device. For example, a Freeview box for a TV or a satnav in a car. If you compare a desktop PC with a Raspberry Pi, you can see that a desktop PC is not suitable as a component to build into small devices, unlike a Raspberry Pi. A standard PC does not normally have any GPIO interfaces fitted either! A Raspberry Pi is much more versatile in this regard than a desktop PC.

Real-time applications

A lot of real-world control applications are said to function in “real time”. For real-time applications, it is often necessary to be able to read inputs, process them and produce outputs thousands of times a second. The rate of this processing has to be predictable – one calculation in a fixed timeframe. For example, a CD has a sample rate of 44.1 kHz – that means that for each data sample point, you only have 1/44100 seconds (22.6 μ S) to do all the processing! In a multitasking operating system, such as Linux on the Raspberry Pi, you cannot guarantee that your program will have full control of the CPU during those few microseconds – the operating system may be busy, communicating on the network port, for example.

When there is an irregular sample rate on input data, this is called “**jitter**”. The predictable timing accuracy that is required needs either dedicated hardware or special real-time operating systems and low-level programming languages, such as C or Assembler. The operating systems currently available on the Raspberry Pi are not really suitable for real-time applications. Fortunately, the Raspberry Pi does include a C compiler (called GCC) if you want to learn how to write a lower-level program.

Don't worry, though, if you had your heart set on creating some real-time applications. There is another small electronics prototyping platform, called an **Arduino**, which contains a programmable microcontroller suitable for real-time applications. It is quite easy to use a Raspberry Pi and an Arduino board together. An Arduino can be used for the high-speed, real-time parts of the design and a Raspberry Pi can run a higher-level GUI or web application that controls the Arduino.

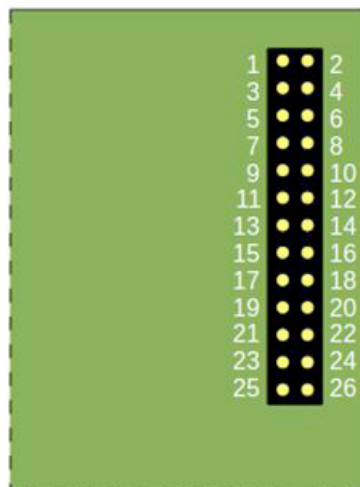
Although the Raspberry Pi can run fast I/O applications and the Arduino can use web applications, this is not what they are best suited for. Remember: always use the right tools for the job! There is an example of communicating between an Arduino and a Raspberry Pi later in this guide.

For more information about Arduino boards, see <http://www.arduino.cc>

GPIO hardware interfaces on the Raspberry Pi

There are several types of interface pins on the Raspberry Pi. They can be configured and used for lots of applications. Note that because the pins on the Raspberry Pi board are connected straight into the **system on a chip** (SOC), it is quite easy to damage your Raspberry Pi or SD card if you are not careful. Make sure you only use **3.3V** on the pins, **not 5V**. For this reason, it is recommended that you use an interface board, such as the **Gertboard**, between the Raspberry Pi and any circuits you build.

DNC = Do Not Connect.
These pins are reserved
for future use.



Pin	Function
1	3.3V
2	5V
4	DNC
6	0V
9	DNC
14	DNC
17	DNC
20	DNC
25	DNC

The maximum permitted current draw from the 3.3V pin is 50mA. The maximum current draw on the 5V pin depends on your power supply – you must leave enough for the Raspberry Pi to run! Pins not listed in the table above are described by type in the sections below. Note that some pins can be configured for more than one type of interface.

GPIO board pins

There are 17 pins available to operate in GPIO mode, configurable as either inputs or outputs. They carry just one bit of digital data.

High = 3.3V

Low = 0V

Board pin	BCM GPIO number
3*	0
5*	1
7	4
8	14
10	15
11	17
12	18
13	21
15	22

Board pin	BCM GPIO number
16	23
18	24
19	10
21	9
22	25
23	11
24	8
26	7

* Note that these pins have a 1.8k pull-up resistor on the Raspberry Pi board.

Inter-Integrated Circuit (I²C)

I²C is an interface on which you can connect multiple I²C slave devices. The Raspberry Pi acts as the master on the bus.

Board pin	BCM GPIO number	Function	Description
3*	0	SDA	Data
5*	1	SCL	Clock

Serial Peripheral Interface (SPI)

SPI is an interface on which you can connect multiple SPI slave devices. The Raspberry Pi can only act as the master on the bus.

There are five pins available to connect devices to the Raspberry Pi using SPI:


Board pin	BCM GPIO number	Function	Description
19	10	MOSI	Master Out, Slave In
21	9	MISO	Master In, Slave Out
23	11	SCLK	Serial Clock
24	8	CE0	Channel Enable 0. Also known as Slave Select (SS)
26	7	CE1	Channel Enable 1. Also known as Slave Select (SS)

Universal Asynchronous Receiver/Transmitter (UART)

The UART is a serial bus connection. Note that these pins run at 3.3V and the RS232 specification is for 12V. If you connect this to a RS232 serial device, you could potentially damage your Raspberry Pi. Please be careful!

Board pin	BCM GPIO number	Function	Description
8	14	TX	Transmit
10	15	RX	Receive

Notes:

**Tip...**

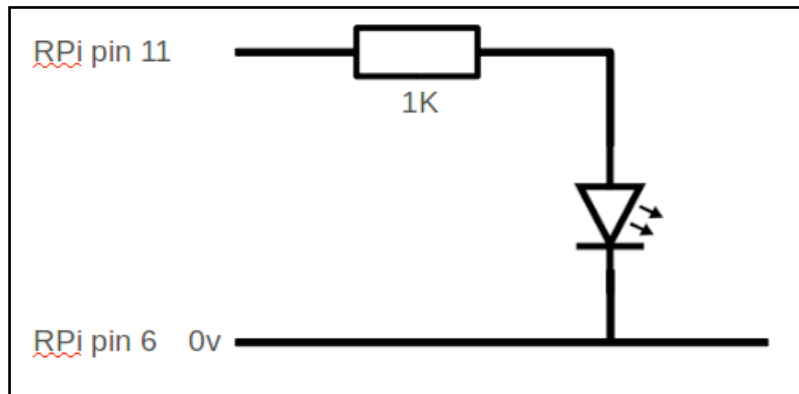
Please be careful with the UART connection! If you use too much current then you could easily break something!

LED circuit

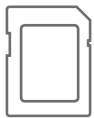
Now we know all about data and the Raspberry Pi's input/output options, let's get on with building something!

This is about the simplest circuit you can build to test the GPIO outputs of a Raspberry Pi. This circuit contains just two components: a 1k resistor and an LED (Light Emitting Diode). The resistor is used to limit the current that flows out of the Raspberry Pi and into the LED. If there is too much current, you could break something!

LED circuit experiment wiring diagram.



Note that you need to connect the LED up the correct way round. The flat side of the LED denotes the negative side; the longer leg denotes the positive side of the LED. The Gertboard already has LEDs wired up exactly like this on some channels.



The following Python program will let you switch the LED on and off. To complete this exercise you will need the Raspberry Pi GPIO modules. You can either install these from the Raspberry Pi SD card or download them from <http://pypi.python.org/pypi/RPi.GPIO/>

```
import RPi.GPIO as GPIO

# set up pin 11 to output
GPIO.setup(11, GPIO.OUT)

state = False
while 1:
    GPIO.output(11, state)
    command = input("Press return to switch the LED on/off or
'Q' to quit: ")
    if command.strip().upper().startswith('Q'):
        break
    state = not state
```

Note that this Python script must be run with superuser privileges (as root). You can do this by running your program from the command line and putting “sudo” in front of the command you are typing. For example: “sudo python led.py”.

Notes:

Tip...

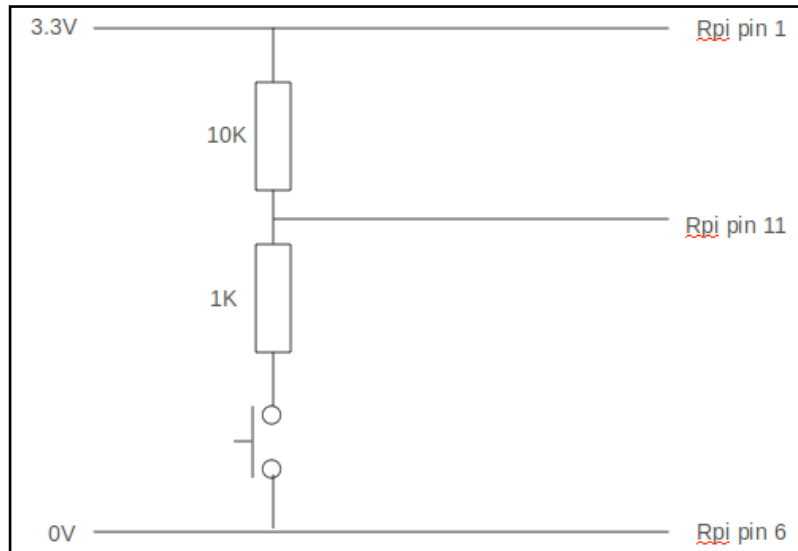
Please be careful:
If you use too
much current
then you could
easily break
something!

Push-button circuit

Notes:

This is the about the simplest circuit you can use to test GPIO inputs with your Raspberry Pi. The 10k resistor is what is known as a “pull-up” resistor – that means that the input will be pulled high (to 3.3V) when the button is not pressed. When you press the button, it connects the input to 0V via the 1k resistor, sending the input low. The 1k resistor is present to protect your Raspberry Pi in case you accidentally set it up as an output instead of an input. The Gertboard has some channels wired up like this circuit.

Push-button circuit experiment wiring diagram.



On the next page is an example of some Python code that monitors a push button. We use the tasking features of Python to create a class that monitors a push button to demonstrate how we might use multitasking. This is so that we do not miss the button press while the program is busy doing other things. You will probably notice that this is similar to checking for events when using PyGame.

When using the Python RPi.GPIO module, LOW = False and HIGH = True. As in the previous example, this program must be run as root by putting “sudo” in front of the Python command.

```
import threading
import time
import RPi.GPIO as GPIO

class Button(threading.Thread):
    """A Thread that monitors a GPIO button"""

    def __init__(self, channel):
        threading.Thread.__init__(self)
        self._pressed = False
        self.channel = channel

    # set up pin as input
    GPIO.setup(self.channel, GPIO.IN)

    # terminate this thread when main program finishes
    self.daemon = True
```

```
# start thread running
self.start()

def pressed(self):
    if self._pressed:
        # clear the pressed flag now we have detected it
        self._pressed = False
        return True
    else:
        return False

def run(self):
    previous = None
    while 1:
        # read gpio channel
        current = GPIO.input(self.channel)
        time.sleep(0.01) # wait 10 ms

        # detect change from 1 to 0 (a button press)
        if current == False and previous == True:
            self._pressed = True

            # wait for flag to be cleared
            while self._pressed:
                time.sleep(0.05) # wait 50 ms

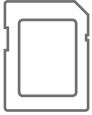
            previous = current

def onButtonPress():
    print('Button has been pressed!')

# create a button thread for a button on pin 11
button = Button(11)

while True:
    # ask for a name and say hello
    name = input('Enter a name (or Q to quit): ')
    if name.upper() == ('Q'):
        break
    print('Hello', name)

    # check if button has been pressed
    if button.pressed():
        onButtonPress()
```



To complete this exercise you will need the Python Serial Port Extension. You can either install this from the Raspberry Pi SD card or download them from <http://pypi.python.org/pypi/pyserial/>. You will also need the Debian package called “arduino”, in order to install the Arduino development IDE. You can download this from the Debian website.

Obviously, you will also need an Arduino board.

Arduino is an open-source electronics prototyping platform based on flexible, easy-to-use hardware and software. An Arduino board connected to a Raspberry Pi is a very useful and powerful combination. You can learn more about the Arduino platform at the website <http://www.arduino.cc/>

This example uses an Arduino board connected to a Raspberry Pi using a USB cable. You do not have to build any circuits to make this program work – that is left up to your imagination. This program is very simple; it asks for a character on your Raspberry Pi, and then sends it to the Arduino. The Arduino responds by returning the character and its ASCII code. Finally, the response is printed on the screen of the Raspberry Pi.

Notes:

Code for the Arduino:

Notes:

```
// set up the serial connection speed
void setup()
{
    Serial.begin(9600);
}

void loop()
{
    int inByte;

    if (Serial.available() > 0)
    {
        // read data from the Raspberry Pi
        inByte = Serial.read();

        // send data to the Raspberry Pi
        Serial.write(inByte);
        Serial.print(" = ");
        Serial.println(inByte);
    }
}
```

Python code for the Raspberry Pi:

```
import serial

# set up the serial connection speed
ser = serial.Serial('/dev/ttyACM0', 9600)

# main loop
while 1:
    c = input('Enter a char: ')
    if len(c) == 1:
        # send data to the Arduino
        ser.write(c.encode())

        # receive data from the Arduino
        response = ser.readline()

        print(response.decode().strip())
```

The operating system on your Raspberry Pi is a version of Linux. In all probability, it looks a bit like Windows, or – more likely – like the Mac OS. It has a graphical user interface (GUI) that allows you to interact with your folders and files by double-clicking, right- or left-clicking, or dragging and dropping. If you want to open a program, you look for it on a menu called something like “Applications”.

That’s all very well, but there is another way to interact with Linux: using the **command line interface (CLI)**. With the CLI there are no images; nothing to click on. To get your computer to do something, you must type a properly constructed text command. Often, you’ll only know that your command has been successfully executed because your computer won’t respond. If it does respond, something has gone wrong and it’s giving you an error message.

That sounds like hard work. Do we really want to bother with it? Yes, we do! For a start, by default, some versions of Linux boot directly into the CLI. To open the GUI, with all its lovely windows and menus, you then need to type the command “**startx**” and press Return. But that’s by no means the only reason to get to grips with the CLI.

With the command line you can do things that you can’t do using the GUI and you can do things easily that are difficult in the GUI. Compared with the graphical interface, the command line hardly uses any processing or graphics power, so it’s great if you have heavy work for your computer to do. And, if you’re connecting remotely to a Raspberry Pi that doesn’t have its own monitor, then you’ll have to use the CLI.

If you’re new to Linux, you may not be familiar with the command line in general and with Linux commands in particular (they’re sometimes similar to their Windows equivalents, but often they’re completely different). This chapter is a quick introduction to the magic of the command line. It contains everything you need to get started.

Notes:

Lesson 6.1: Commands are just programs

When you type into the terminal you are running programs. Most of the commands listed here run programs that give you the ability to command the system to do something. When you add programs to your Raspberry Pi, you will be able to run more commands.

If you enter a command and it doesn't work, it may well mean that the program isn't installed yet. For instance, if you try to create a new text file called "My Linux Commands" using the Nano text editor, you would type:

```
sudo nano "My Linux Commands"
```

But it might be that, instead of opening the new text file, your system returns the error message:

```
bash: nano: command not found
```

This just means that you have not installed Nano yet. However, most of the commands described on this page are the kind of "housekeeping" commands that come as default, so you shouldn't have this problem. Each program has an original author who is acknowledged at the bottom of the main page.

Don't worry too much about the actual commands used there – I will explain everything as we go along.

The Bash shell

To use the CLI, you need to know commands. The default command set is contained in the "**shell**" you are using. There are lots of shells out there, but the one on this Raspberry Pi is called "**Bash**". Bash is the default command line shell on most versions of Linux, as well as Mac OS, so it's well worth learning.

This chapter covers many of the most common and useful commands. If you can't find the command you need, try looking online. There are lots of good guides on the internet for using the Linux command line.

If you know the command you want to use, but don't know exactly how to use it – for instance, you don't know exactly the right "**syntax**" – you can use the "**man**" (manual) command in Linux. At the command line you could type:

```
$ man <command>
```

man

Displays information about the target, sourced from online reference manuals.

Notes:



Tip...

The "syntax" of a command means the way it should be written to make it work. That means the order of the words, as well as any important punctuation that is also necessary.

Or, you could use the “info” command:

Notes:

```
$ info <command>
```

info

Displays online documentation about the target.

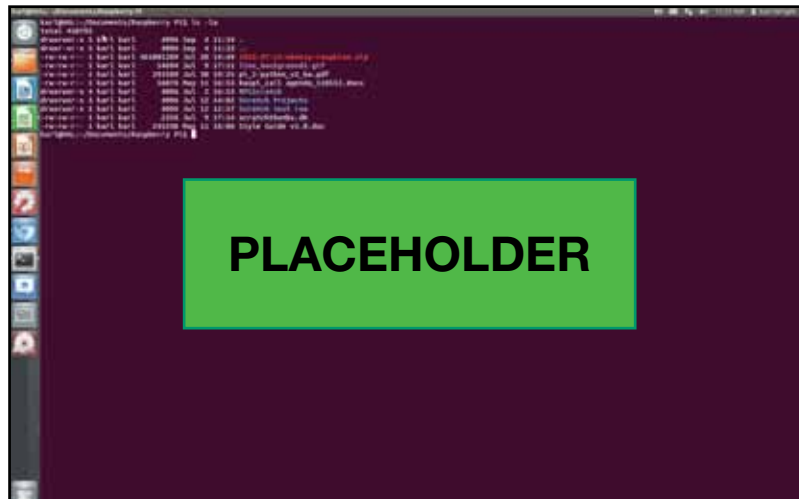
In the examples above, substitute <command> with the command you want to find out more about. A word of warning: the guidance given by “man” is sometimes a bit formal and very, very detailed. You could almost say, “If you think you know a command then go to the command’s man page in order to find out that you don’t really.”

To find out more about Bash, take a look at its Wikipedia entry:

http://en.wikipedia.org/wiki/Bash_%28Unix_shell%29

Typing “man apt-get” into the command line returned all this information on the “apt-get” command and how to use it.

“Man” is short for manual.



Lesson 6.2: Command syntax and file structure

Notes:

Commands take the form:

<Command> | <Switches> | <Parameters> | <Target>

In guides, such as this one, the brackets “<” and “>” are often used to indicate the place a command would take in a string of text being typed into the CLI. The horizontal line “|” is used to denote “or”. Confused? Okay, let’s look at the example above. We would read that as saying that commands can be used by themselves OR they can be used with:

- **Switches:** single letters, preceded by a hyphen, that adjust what the command does
- **Parameters:** things that the command needs to know in order to work
- **A target:** the thing (such as a file) that the command will be applied to

Let’s look at an example. We’ll start with the “**ls**” command, which you can use to see a list of a folder’s contents.

```
ls -l /home/brian
```

Command Switch Target

This command tells the command line to list, in long format, the contents of the directory “/home/brian”. The *command* is “ls”; the *switch* “-l” tells Linux that you want the list in long format; and the *target* of this command is the directory “/home/brian”. In this example, there are no *parameters*, so that part of the command is just skipped.

ls

Lists the contents of directories.

For a slightly more detailed example, let’s look at the “**mount**” command, which tells Linux to incorporate a new file system (such as a CD or DVD) into its own file system structure so that you can browse it.

```
mount -F smbfs //workgroup;fred:foo@192.168.1.99/homes /mnt/net
```

Command Switch Parameters Target

This command tells the operating system to use the username “fred” and password “foo” (parameters) to make the shared drive called “homes” on the Windows server at 192.168.1.99 (parameter) appear in the directory tree at the point “/mnt/net” (target) using the Server Message Block Filing System (the -F switch).

mount

Makes a file structure available at a new location.

Navigating the file system using “cd”

The file system in Linux is hierarchical with nested directories (often called “**folders**”) in a “**tree**”. The top of the directory structure is denoted by the symbol “/”, and directories underneath “/” are referred to using “**paths**”, just like URLs in a web browser.

To go to a particular place in the directory structure, you use the command “**cd**”, which stands for “change directory”, followed by its location in the tree. For instance, the following command will take you to Brian’s “Documents” folder in his home directory.

```
cd /home/brian/Documents
```

If you want to move up the directory tree, use the command:

```
cd ..
```

For example, if you’re working in the folder “/home/brian/Documents/Project”, the command “cd ..” would take you back to “/home/brian/Documents”. Make sure you leave a space between the letters “cd” and the two dots, otherwise the command won’t work.

To find out more about the “cd” command see its Wikipedia entry:

http://en.wikipedia.org/wiki/Cd_%28command%29

cd

Changes the current working directory.

Notes:



Tip...

The names of files and directories (folders) are case-sensitive in Linux, so the file “my_stuff” is different from the file “My_stuff”.

Listing the files and folders in a particular place

Notes:

The “ls” command lists all contents of directory you’re working in. There are a range of switches you can use with “ls” to make it display exactly the files you’re interested in and all the useful information about those files. We’ve listed a few of those switches here:

- “ls”, the command with no switches, lists all the file in the current directory.
- “ls -l” lists the files and displays the long version of the information about each file or directory. Output may be colour-coded depending on the terminal preferences that you have set.
- “ls -R” asks for a “recursive” list – that is, a list including the contents of sub-directories as well as this directory.
- “ls -A” forces the system to show “hidden” files. Hidden files have names that start with a dot, which won’t usually be visible when doing a normal directory list.

Let’s look at two examples. In the first, we use the plain “ls” command with no switches.

```
$ ls
```

```
An_Gott_und_meine_Mutter.mid Domestic Programming Test.mid  
An_Gott_und_meine_Mutter.mscz Engineering Quantum Physics Tutoring  
appliances FluidR3_GM.ins School Windaz
```

In the second, we use “ls -l”. This displays the files in long format, telling you, among other things, the size, owner and security setting on each file.

```
$ ls -l
```

```
total 336  
-rw-rw-r--. 1 brian brian 2429 Apr 2 20:27 An_Gott_und_meine_Mutter.mid  
-rw-rw-r--. 1 brian brian 4085 Apr 2 19:52 An_Gott_und_meine_Mutter.mscz  
drwxrwxr-x. 4 brian brian 4096 Apr 2 20:38 appliances  
-rw-rw-r--. 1 brian brian 10919 Apr 2 19:52 brotplot.odt
```

Don’t worry, we’ll go into the details of this information later.

To find out more about the “ls” command, take a look at its Wikipedia Entry:

<http://en.wikipedia.org/wiki/Ls>

Change ownership of directories and files

Notes:

In the long List above, the files are owned by Brian and are also in the Brian group. What if we wanted to change the ownership of some of those files? Then we would use the “**chown**” command.

We'll use the file “*brotplot.odt*” as an example. Let's say we want to make Fred the file's owner. We would use the command:

```
$ chown fred brotplot.odt
```

If we run the command “ls -l” again, we will now see the result:

```
-rw-rw-r--. 1 fred brian 10919 Apr 2 19:52 brotplot.odt
```

The file is still in group Brian but is now owned by Fred. But what if we want to make the user “Foo” the owner of the file and also move the file to the “foo” group. We'd use the command:

```
$ chown foo:foo brotplot.odt
```

If we use “ls -l” again to see the long-format name of our file, we will see:

```
-rw-rw-r--. 1 foo foo 10919 Apr 2 19:52 brotplot.odt
```

To find out more about “chown” command see its Wikipedia entry:
<http://en.wikipedia.org/wiki/Chown>

chown

Changes the ownership of one of more files.

While we're at it, take a look at “chgrp” to see how it compares:
<http://en.wikipedia.org/wiki/Chgrp>

chgrp

Changes the group of one of more files.

Change access to directories and files

Notes:

When you use the “-l” switch to see the long list of information a directory’s contents, the results you see are all preceded by a string of letters and dashes. For instance:

```
-rw-rw-r--. 1 fred students 10919 Apr 2 19:52 brotplot.odt
```

The first character in the sequence, in this case a dash, tells us what the object is. The most commonly-used characters are:

- d** Indicates a directory
- l** Indicates a link
- Indicates a file

Our example above starts with a dash, so we know it’s a file. In this case, it’s actually a word processor file.

The nine characters that follow, describe access rights for the owner (user), group and the world (everyone else who may be able to access the file), in that order, each with three characters. The characters used here are:

- r** Read access
- w** Write access
- x** Permission to execute the program
- No access of this type

So, if all three groups had all three types of access, we would see “rwxrwxrwx”. However, whenever someone doesn’t have an access type, its letter is replaced with a dash.

If we again look at our example above, we can see that:

- Brian has read and write access to the file.
- Members of the group called “students” also have read and write access to the file.
- Everyone else (the “world”) has only read access to the file.

The “x” flag is not listed at all. Remember, x stands for the ability to execute a file. If the x flag is not set then the file will not run as a program.

But what if we want to change the file’s permissions? There are various ways to use the “**chmod**” command to do this, but probably the easiest is to use these groups:

- u** for user/owner
- g** for group
- o** for other (everyone else)
- a** for all three

Now, let's try out the "chmod" command:

```
chmod o+w brotplot.odt
```

This means add write access for the "other" group to "brotplot.odt". Add is indicated by the "plus" sign.

```
chmod a-r brotplot.odt
```

This means remove read access for the all three (user, group and other) from "brotplot.odt". Remove is indicated by the "minus" sign.

```
chmod u+x brotplot.odt
```

This means add execute access for the user/owner to "brotplot.odt".

To find out more about the "chmod" command see its Wikipedia entry:

<http://en.wikipedia.org/wiki/Chmod>

chmod

Changes the access mode (permissions) of one or more files.

Lesson 6.3: The superuser

Notes:

The “**sudo**” command introduces the “superuser” or “root user”. The term “root” is the name for the main administrator in a “Unix-like” system, such as Linux. There are many commands that only the root user can run.

Depending on the version of Linux you are using, you will either have to log on as the root user or prefix your command with “sudo”. The default Debian distribution of Linux, for instance, has no root password set. So you will have to use the “sudo” command.

By using “sudo”, you’re saying, “Do the following command as the root user.” When you do this, you will be asked for your password and, if you have the system permissions of a root user (commonly called “being in the admin group”), then the command will be run. If you don’t have root permissions, you will get an error message.

Let’s look at the example of adding a new user to your system, something that only the root user can do. On Debian, the command would look like this:

```
sudo adduser brian
```

This will start a script allowing you to set up a new user called “brian”. If you ever try to do something in the command line and get the error message, “Only root can ...” then try the sudo command – it may fix your problem.

To find out more about “sudo” command, see its Wikipedia entry:

<http://en.wikipedia.org/wiki/Sudo>

sudo

If you have the appropriate permissions, execute the following command as the superuser.

Becoming the admin user

There is another way of doing things as root user and that is by using the “**su**” command, which stands for “substitute user” (or superuser). Invoking the “su” command means “become the root user”.

As before, you’ll be asked to enter your password to make sure that you have superuser rights. Once you have successfully authenticated with your password, all commands run as the superuser.

In some systems, however, you won’t be able to use the “su” command because, by default, the root user isn’t enabled. In this case, you will have to enable the root user, using this command:

```
passwd root
```

You will be asked to enter and then confirm a new root password. Assuming you can manage to enter the same password twice, you now have a root user and you can issue the command:

```
su -
```

When you enter passwords in Unix-like systems, the prompt remains blank: no blobs or asterisks stand in for the characters in your password. Don't be put off; the password is being entered nevertheless.

Finally, a word of warning: if you habitually do everything as the root user, eventually you will do something both educational and disastrous. For example, you could invoke:

```
rm -r -f /*
```

This will delete all the files in the whole system.

Or you might absent-mindedly type:

```
$ rm -r -f ./*
```

That would just delete the files in the current directory. And you do not usually get warnings as root user (other than this one). The first thing you'll know about your error is when all your files have suddenly vanished.

Only use root privileges when you really need them.

To find out more about the "su" command, see its Wikipedia entry:
http://en.wikipedia.org/wiki/Su_%28Unix%29

su

Become a superuser.

passwd

Create or change a password associated with the identified user.

Notes:

 **Tip...**

Don't ever use these commands when operating as a superuser!

Lesson 6.4: Creating and destroying files and directories

Notes:

Finally, let's look at creating and destroying file and directories, as well as taking a closer look at the mount command.

Create a new empty file

To create a new empty file, go to where you want the file to be and type:

```
$ touch <filename>
```

The “**touch**” command actually updates a file’s “last accessed” time to the current time and date, but if such a file doesn’t already exist then it will create a new file of that name, with a file size of 0.

To find out more about the “touch” command, see its Wikipedia entry:
[http://en.wikipedia.org/wiki/Touch_\(Unix\)](http://en.wikipedia.org/wiki/Touch_(Unix))

touch

Update the named file's access time and modification time (and dates) to the current time and date. If a file doesn't exist then a new file will be created with a file size of 0.

Create a new empty directory

To create a new directory, go to where you want the directory to be and type:

```
$ mkdir <directory name>
```

To find out more about “mkdir” command, see its Wikipedia entry:
<http://en.wikipedia.org/wiki/Mkdir>

mkdir

Create one or more directories in the current location.

Remove a file

To delete a file, go that file's location and type:

```
$ rm <filename>
```

To find out more about the “rm” command, see its Wikipedia entry:
http://en.wikipedia.org/wiki/Rm_%28Unix%29

rm

Remove (delete) one or more files in the current location.

Remove a directory

To delete a directory, go its location and type:

```
$ rm -r -f <directory name>
```

This uses the “rm” command we used above, but the extra switches tell it to remove the directory, all its contents and also any sub-directories and their contents.

The “-r” means “recursive”, which – in the case of a directory – removes the entire directory and all its contents, including sub-directories. Do be very sure that you mean this before using it – there’s no “undo” option!

The “-f” switch indicates that the action is “forced” – that is, the program will remove write-protected files without prompting. This is also dangerous, so be very careful when using this as well.

A safer, but less powerful, option is to use the “rmdir” command:

```
$ rmdir <directory name>
```

This will also delete directories, but only if they’re empty.

rmdir

Remove (delete) one or more directories in the current location, provided they are empty.

Connect to a device or filing system

The “mount” command allows you to connect a Unix system to external devices. There is no “C” drive, as in Windows. What happens in Linux is that a device is “mounted” somewhere in the filing system. When you navigate to that place, the items offered by the device will appear at that point.

This is a complex command. The switches, parameters and target of the mount command will vary according to the protocol of the system being mounted. Some things will “auto-mount”. This is why, when you plug an SD card into a modern Linux system, the filing system will automatically pick it up.

Manual mounting requires a “mount point”. That means a directory that will be filled with the mounted device when it is mounted. Often, this is in the directory “/mnt/” somewhere. Generally, before mounting new media, you must first ensure that there is a mount point. If there isn’t, then you must create a directory at the point needed: for instance...

```
$ mkdir /mnt/netfolder
```

You must also make sure that you use the necessary switches, parameters and directories. For instance, from the previous exercise:

Notes:

```
mount -F smbfs //workgroup;fred:foo@192.168.1.99/homes /mnt/net
```

This, as we learned earlier, tells the system to mount the shared drive called “homes” on the Windows server at 192.168.1.99 , in the directory tree at the point “/mnt/net”, using the Server Message Block Filing System (the -F switch).

To find out more about the “mount” command, see its Wikipedia entry:
http://en.wikipedia.org/wiki/Mount_%28Unix%29

Further learning

The commands presented here are just a small selection of all the commands available on your system. You can find a comprehensive list of Linux commands on the O'Reilly website:

<http://www.oreillynet.com/linux/cmd>

Lesson 6.5: Remote access to the Raspberry Pi

Notes:

OpenSSH is an application that allows you to securely access Linux systems remotely over the network. You can use OpenSSH simply for secure file sharing. But it also allows you to log on to a system and control it over the network, even using the GUI, just as if you were sat in front of it.

The default installation of Linux on your Raspberry Pi should have “**SSH daemon**” running. This means that your Raspberry Pi is listening on port 22 for a remote computer asking to make a connection to it. In your case, this will probably mean your normal desktop or laptop computer.

Running a remote CLI

To connect to your Raspberry Pi, you will need an SSH client program. Linux and the Mac OS already have these installed. For Windows, you can download PuTTY (<http://www.chiark.greenend.org.uk/~sgtatham/putty>). Refer to the manual of your chosen SSH client program for more information on how to install and use it.

In order to make a successful connection, you must have port 22 open on both your remote host and the Raspberry Pi. You will also need to set up a suitable user on the RPi (as we did earlier in this chapter).

If you are using a Linux or Mac client, simply enter this at the command line:

```
ssh <IP address of RPi> -l <username on RPi>
```

<IP address of RPi> should be replaced with the IP address of your Raspberry Pi, and <username on RPi> should be replaced with the username of your Raspberry Pi, as you set up previously. The character between them is a lower-case “L”.

You should see output similar to this:

```
[brian@fc16toshiba ~]$ ssh -X 192.168.1.104 -l brian
The authenticity of host '192.168.1.104 (192.168.1.104)' can't be established.
RSA key fingerprint is 26:a4:a1:ab:c2:ff:50:99:d7:e1:49:6e:f2:90:fb:90.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.1.104' (RSA) to the list of known hosts.
brian@192.168.1.104's password:
Linux raspberrypi 3.1.9+ #9 Mon Apr 9 20:50:36 BST 2012 armv6l
The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.

brian@raspberrypi:~$
```

At this point you are up and running. Everything you type is actually happening on your Raspberry Pi.

Running a remote GUI

To remotely connect to your Raspberry Pi and use its GUI you will need to install an “**X server**” on your machine. Linux has an X server built in. For Windows, try Xming (<http://www.straightrunning.com/XmingNotes>) or Cygwin (<http://www.cygwin.com>) – you can download both for free. For the Mac OS, go to “Shell > New Remote Connection” and choose “Secure Shell (ssh)”.

The following command uses the X switch to tell SSH to send the X commands to the X server on your host. Enter the following:

```
ssh -X <IP address of RPi> -l <username on RPi>
```

What is an X server?

A Linux computer (or other Unix-like system) running a GUI is almost certainly running “X” and also an X server. The X sends commands to the X Server about what kind of things to put on the screen, and the X server does it.

This means that the GUI and the X server are separate. This also means that you can run a program on the Raspberry Pi and have the graphical output appear on the screen of the X server somewhere else on the network. This removes a large amount of processor demand from the Raspberry Pi. And it also means that you can just plug your Raspberry Pi into your network, rather than giving it its own monitor, mouse and keyboard.

Open a remote browser

Notes:

Let's see what happens when I try to run a web browser remotely. The web browser on the Raspberry Pi is called **Midori**, so this is what we need to enter from the prompt after we connect remotely:

```
brian@raspberrypi:~$ midori &  
[1] 5773
```

We need to include the “&” symbol after the program name because this tells the server to launch it as a separate process, which means we get our command line back.

We get the response “[1] 5773”, which is the process number of the Midori program now running.

Sometimes you will get GTK errors, which report library shortcomings on host (the RPi) or server (your X server) but the process is pretty robust.

*Midori running in Remote
Gnome 3 UI using the
Fedora 16 X Server.*



If you have worked through this manual you will have a good understanding of the basics of computer science and you will be able to write a computer program to solve simple problems. This bears repeating: you can write a set of instructions to make a computer do something. This simple statement embraces a set of powerful thinking skills that include logical reasoning, problem solving, algorithm design and much more. As of now, you are no longer just a consumer of digital products – you are a creator! This is something of which to be proud.

We hope that this brief introduction has shown you how useful, exciting and fun computing is as a hobby or career. If, at any time while following this guide, you grinned to yourself or you shouted downstairs, “Come and see what I’ve made!” then we have done our job – this is, after all, why the Raspberry Pi was made. If you don’t go any further with your Raspberry Pi journey then you at least know what all the fuss is about and that it’s not as hard as people think. It is quite probable, however, that you have caught the computing bug and would like to learn more.

So - what now? There really is no right or wrong way to continue your exploration of the Raspberry Pi. It all depends what interests you. It could be as simple as getting better at your chosen programming language, or improving your Linux command line skills. Small projects, such as setting up a home media server or writing a simple game, are also a good way to start. At some point, most people will want to have a go at interfacing the Raspberry Pi with the outside world (this is one of its strong points) using a breakout board such as the Gertboard, or by making their own interface. Again, a simple project such as monitoring temperatures or controlling a cheap robot is the place to start.

As your understanding of the Raspberry Pi grows and your programming skills improve, you will find that your projects get more and more complex, and that the Raspberry Pi becomes a serious tool for experimentation and creativity. Keep practising – one of the beautiful things about programming is that you can take a blank text file and create something that previously only existed in your head. What you make is limited only by your imagination.

Where can I get help, ideas and inspiration?

The best resource to help you continue your computing journey is the web. It’s vast, full of people with large brains, chock full of diverse and arcane knowledge, and unlike this manual it gets updated frequently. To save time we were just going to steal a huge list of links from somewhere, paste them here and make ourselves scarce. But our editors told us off. Here then are our links with a few comments¹. It’s only a start, but should save you some time trawling the web.

¹ *Disclaimer: if a resource is mentioned here, it does not mean it is the best in class. Likewise, if a resource isn’t mentioned, it doesn’t mean that it’s no good. We have no agenda apart from getting people into computing and we have no affiliation with any of the sites or people mentioned. We do rather like the Raspberry Pi Foundation, though.*

It concentrates on programming because the Raspberry Pi was made to encourage a new generation of programmers. While there is much more to computing than programming, it is a very good place to start.

General resources and help

The first place to look is the official Raspberry Pi website and forum (<http://raspberrypi.org>). There is a lot going on in the forums and there are subforums for specific topics such as GPIO, programming, operating systems and education. The members are a friendly bunch who collectively have a huge amount of experience and expertise. They would be happy to answer any questions, help you with problems and point you in the right direction. If you have a question about your Raspberry Pi or are trying to get something to work then this is probably the best place to start.

The Raspberry Pi wiki at eLinux (http://elinux.org/R-Pi_Hub) is also a great source of information and is constantly updated. It's a wiki, so feel free to add to it and improve it as your knowledge grows and your expertise widens!

Perhaps the most useful thing that you can do is to join some forums (no, not *fora*) and talk to people. Get involved, ask questions, help others. Above all keep on computing and have fun!

Programming

The first question that those new to programming ask is, "What is the best language to learn?" Put simply – there isn't one. Just discussing programming languages would fill another volume (and would make you claw frantically at your keyboard in bored rage, like some deranged man-mantis).

Each language has its advantages, disadvantages and particular uses, and all programmers have their favourites. It can all get a bit, "My dad's better than your dad", which isn't that helpful if you don't know where to start. Ultimately, it's the underlying computational concepts that you learn while programming that actually matter. So our advice at this stage is this:

The first language you learn is unimportant. Pick one and get programming!

[Update: the author of this statement has since had to go into hiding, after he was attacked in his local supermarket by a horde of angry programmers, who hurled turnips at him and called him an "ill-educated homunculus".]

So, pick a language and get started. Unless you have reason not to, then continuing with Python seems sensible. Eventually, you'll learn other languages but by then you will know what you want to learn and why.

Raspberry Pi-specific resources

Liam Fraser has made a series of Raspberry Pi-specific YouTube tutorials, which you can find at <http://goo.gl/MM9hA>

The Magpi, the free Raspberry Pi Users' magazine (<http://themagpi.com>) has some excellent beginners' articles and tutorials.

Python online tutorials, resources and references

The official Python site keeps a huge list of quality resources both for beginners (<http://goo.gl/MMo5G>). It should keep you busy for some time.

Online courses

Free online courses by top universities have taken off recently, and they teach many computing-related topics, such as programming, electronics, artificial intelligence and logic. The courses are pitched at college/university level but the beginner courses could be tackled by younger students.

The courses are typically delivered via video lectures and handouts, with accompanying exercises. Many have definite start and end dates, with homework deadlines and formal exams, although some – such as the Udacity CS101 course – are now 'open' and you can start (and finish) any time. These courses are ideal for people who want to do some more in-depth stuff and would like guided tuition with a specific target.

Currently, the main providers are:

Udacity (<http://udacity.com>)

The *Introduction to Computer Science* (CS101) course needs no prior computing or programming experience. It takes you from nothing to building your own search engine (though we don't think that Google should be too worried!).

Coursera (<http://coursera.org>)

Coursera offers 120 courses across 16 categories. *Computer Science 101* is the obvious beginner's computing course but hasn't started as of October 2012. One advantage is that it has a self-study option where you proceed at your own pace (but you don't get a certificate).

edX (<http://edx.org>)

A collaboration of MIT, Harvard and Berkeley, edX currently offers two introductory computing courses: *CS50x Introduction to Computer Science I* (Harvard) and *6.00x Introduction to Computer Science and Programming* (MIT) due to start in October 2012.

Online practice and tutorials

More informal ways of learning to program online include Codingbat (<http://codingbat.com>) and Codecademy (<http://codecademy.com>). Codingbat is a series of programming challenges that help build strengths in specific areas, such as string handling and logic. Teachers can track student progress, and tools include a useful graph that logs incorrect and partially correct attempts at a solution (this makes it hard to cheat!). It has exercises for Java and Python. Codecademy is a great place for anyone to start programming from scratch. It's user friendly and it tracks your progress. It teaches Javascript but a Python "module" has recently been added.

Less formal online challenges can be a big motivator. These exist for both specific languages, such as the addictive Python Challenge (<http://www.pythonchallenge.com>), and for specific problem-solving, such as the maths-based Project Euler (<http://projecteuler.net>). A full list can be found on the Raspberry Pi forums at <http://goo.gl/n7ej4>.

Beyond online learning

Online practice and exercises are useful but the best motivation to learn to program is to actually make something. This means starting a project with a specific purpose. Another way to improve is to work on projects with other people. See if there is a local Raspberry Pi or Linux user group near you. If you are at school or college join the computer club (or set one up).

Look at other programmers' code and try to understand it. Change it for your own purpose or try and improve it. If it's broken, try and work out why. Don't forget that programming at this level should be fun – extending yourself is one thing but don't get so frustrated that you give up. Talk to people on forums and have a look at how they have solved the problem, there's always more than one way to tackle it.

The end

We hope that you've found this manual useful and that learning to program has been as mind-expanding as it was for the authors when they wrote their first lines of code. Computers are amazing things and being able to control them is an amazing ability.

Keep on programming, hacking, playing, experimenting and creating. And above all, have fun!