

**University of Hull**  
**Department of Computer Science**

**Talking to Hardware with a Raspberry Pi**

Vsn. 1.1 Rob Miles 2013

## Introduction

Welcome to our Raspberry Pi hardware sessions.

Please follow the instructions carefully. If you get the wiring wrong your programs will not work and there is a good chance that you will destroy the delicate circuitry in the Raspberry Pi that you are using.

In this session you will learn a bit about electronics and how to control simple circuits using a Raspberry Pi device. Here are a few conventions used in the text.



This indicates a warning to be careful about this bit. If you get it wrong it might be time to buy a new Raspberry Pi.



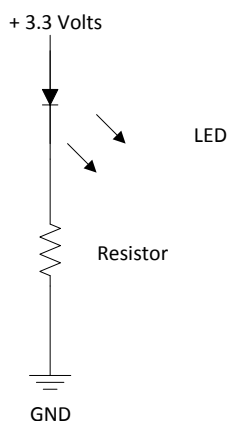
This indicates an activity you should perform in at this point in the text. You may be given precise instructions, or you may have to work something out for yourself.



This indicates something that you may want to think about later.

## Creating a simple circuit

We are going to start by creating a simple circuit to light up an LED. The circuit is as follows:



*Figure 1: A very simple circuit*

The current flows through the LED and causes it to light up. The resistor is there to reduce the flow of current so that that the LED receives just enough to light, but not so much that the LED overheats and fails. The 3.3 volt source is from the Raspberry Pi power supply. You might expect that the voltage you are going to use is 5 volts; after all, that is what the Raspberry Pi power supply produces. However, it turns out that the actual Raspberry Pi computer runs on 3.3 volts to reduce the power consumption of the computer.



Computers work by storing patterns of 0 and 1 and then “twiddling” with them. In some computers the value of 1 is represented by a 5 volt signal, but in the Pi this voltage is reduced to 3.3 volts to reduce power consumption. You might want to think about how this helps. The equation  $\text{Power} = \text{Current} * \text{Voltage}$  might be useful.

## Breadboards and circuits

We are going to create our circuit using a *breadboard*. You can buy these in various shapes and sizes. They can be used to *prototype* a design and prove that it works, before you get the hardware actually manufactured. Alternatively, for very simple projects that you don’t want to mass produce, you can use them to produce the finished product. After all, not many people look inside the box, right?

Breadboards are great for building simple circuits. They provide sockets that you can push wires and component leads into to link them together. Figure 2 shows the breadboard that we are going to use. The neat thing about a breadboard is that the sockets are connected underneath, so you can link things together without needing any (or at least many) wires.

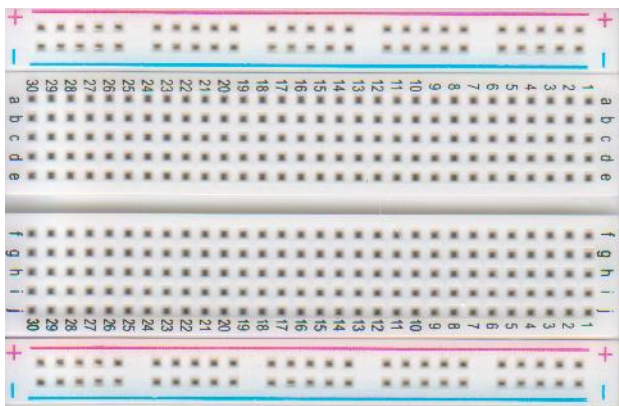


Figure 2: Our empty breadboard

If you look very carefully at the breadboard in Figure 2 you will find that actually it is made up of three parts. The top and bottom rows are separate. This means that the breadboard has a central area for your components and then two connectors along the top and the bottom for delivering power and providing a ground connection.

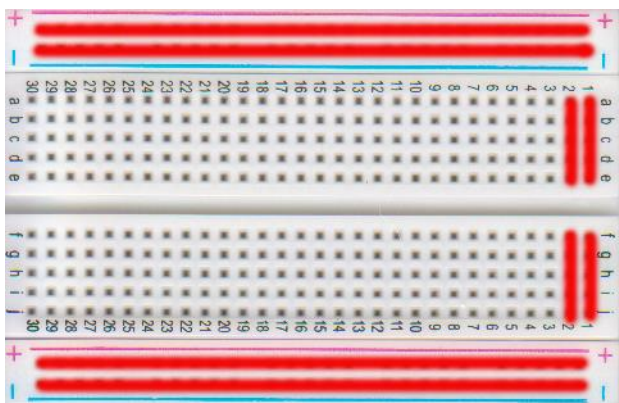


Figure 3: Breadboard connections

Figure 3 shows how the sockets are connected together on the breadboard that we are going to use. The two rows of pins along the top and the bottom are connected horizontally, and each vertical row of pins in the “component” area is connected together, with a gap between the rows. Each of the columns in the central area has a number, and each of the rows has a letter. This means that we can say things like “a 1” and know that this means the socket at the top right hand side of the board.

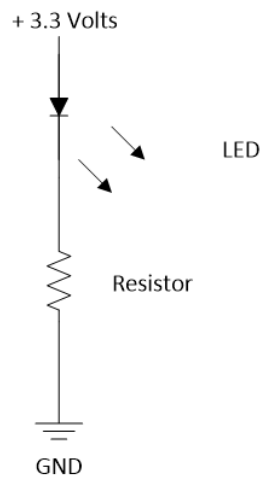
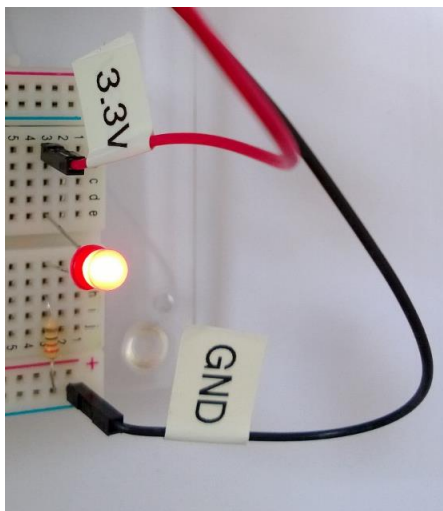


Figure 4: A circuit on a breadboard

Figure 4 shows how we take a circuit and implement it on a breadboard. The resistor makes the connection between the power socket and the LED bridges the gap between the two vertical rows. The two connectors are labelled 3.3V and GND and are connected to the appropriate pins on the Raspberry Pi.

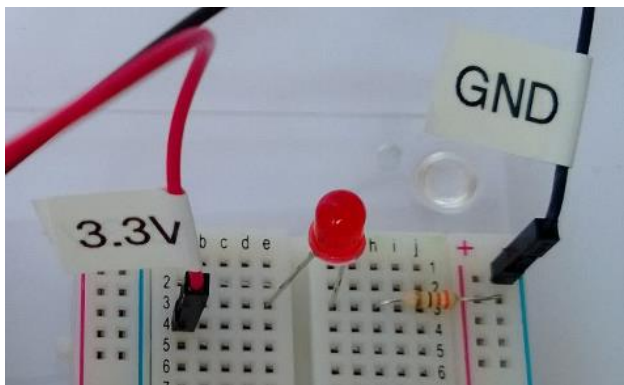


Figure 5: Something wrong here



Figure 5 shows a LED that is not lit up. Can you see what is wrong here?

Of course we would not usually want to just connect a LED to a power supply; we want the Raspberry Pi to control it. And of course we would like to have more than just one LED so that we can signal different conditions.

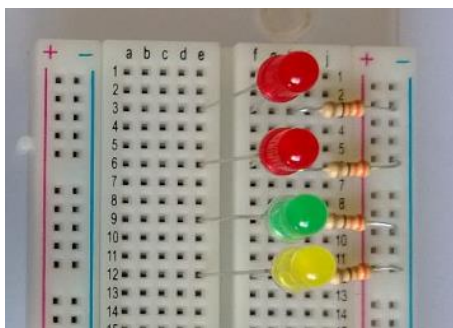


Figure 6: Plenty of LEDs

Because the ground connector runs across the bottom of the breadboard we can connect lots of LEDs to it. Figure 5 shows how this would be arranged. To make life easier, the boards that you will be using have been wired up as above.

## Raspberry Pi Signals

At the moment you have just seen cables with labels sending signals into the breadboard. Now we need to consider what these cables are plugged into.

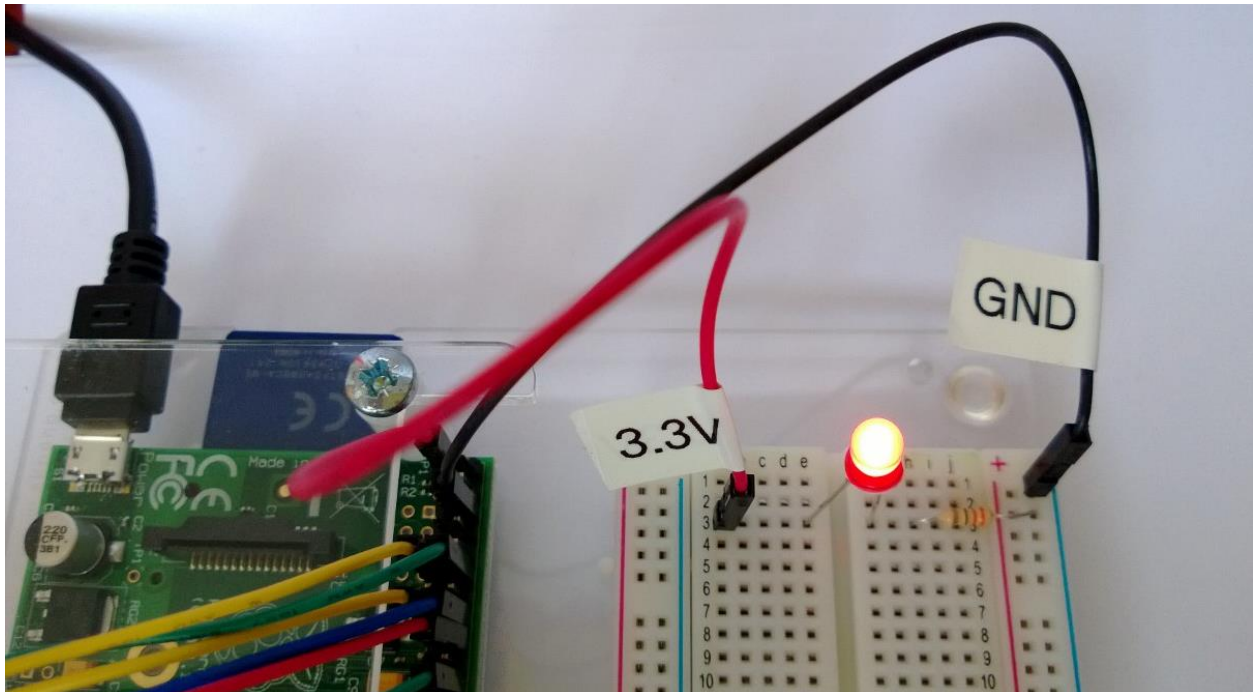


Figure 7: Where the wires go

The Raspberry Pi device exposes a bunch of pins which can be used to connect it to the signals from the outside world. Figure 6 shows the top part of this connector, which is basically a bunch of vertical pins that you can push sockets onto.



Figure 8: Raspberry Pi connectors

Figure 8 shows the actual pins on a “naked” Raspberry Pi. There are 26 pins, but not all of them can be used to connect devices. Some are not connected and others are assigned special functions.



Be careful when connecting things to these pins. It is easy to plug things in the wrong place. Make sure you understand the pin numbering system (coming next) and that you only connect to the pins when the Raspberry Pi is switched off.

Each pin on the connector is identified by a particular number. The numbering might not be how you expect it to be. Rather than numbering along the long side of the connector the pins are numbered along the short side.



2 5v	4	6 GND	8	10	12 GPIO	14	16 GPIO	18 GPIO	20	22 GPIO	24	26
1 3.3V	3	5	7	9	11 GPIO	13	15 GPIO	17	19	21	23	25

Figure 9: Numbered pins

Figure 9 shows how the pins are numbered, and the function of some of them. The pins that are filled in as grey boxes are not available for use by our hardware. I can't really tell you what they do, but I can tell you that if you connect things to them you might have to buy a new Raspberry Pi shortly afterwards.

## Power Supply Pins

Pins 1 and 2 are special, because they are where we can get power from. Pin 1 is connected to the 3.3 volts that the Raspberry Pi CPU uses and Pin 2 is connected to the 5 volt supply coming into the Raspberry Pi itself. You can use these outputs to drive small devices which you might put on the breadboard. Be aware however that the amount of power you can get from the pins is quite limited. If you try to draw too much power you may damage the Raspberry Pi or cause it to shut down for a while if the fuses that protect the power supply overheat.



Unless you really (and I mean really) know what you are doing, you should never use the 5 volt supply. The higher voltage it produces can damage the Raspberry Pi processor. Make sure that anything you use is connected to the 3.3. volt supply.

## General Purpose Input/Output Pins

Pins 11, 12, 15, 16, 18 and 22 can be used as General Purpose Input Output (GPIO) pins. This means that we can write programs that drive outputs (to turn LEDs on and off) and also read inputs (to detect switches). That is what the General Purpose part of the name means. We are just going to use the six pins above, you can actually use more than this if you know what you are doing, but for the size of the breadboard that we are using 6 pins are more than enough.

## Other Pins

If you look at a more detailed description of the connector you will find that you can use the other pins for lots of interesting functions. There are serial connectors that you can use to send data to other devices and lots of other fun stuff. However, just at the start we are going to use the pins above.

## Wires and Labels

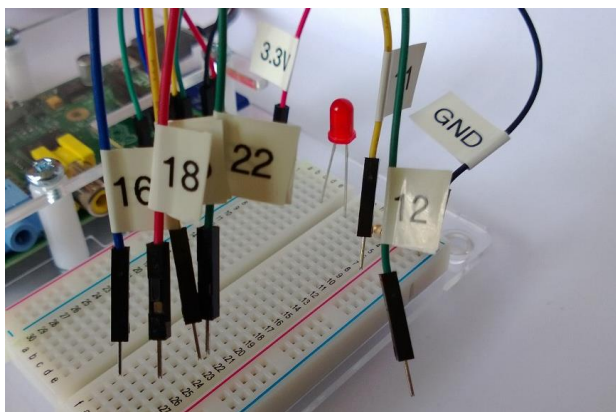


Figure 10: Labelled Pins

To make life easier for you we have actually labelled each of the pins so that you know which is which. You should not have to touch the connector, just use the labelled pin. The colours of the

wires themselves are not particularly important, except that the 3.3 volt cable is red and the GND cable is black.



Please be gentle with the pins and the wires. Always pull the plug out by gripping the black plastic bit and not by tugging the wire.

## Testing your Board

The first thing you need to do is prove that all the LEDs work correctly. There is nothing worse than spending ages trying to work out why a program doesn't make the hardware work, if the hardware is broken.

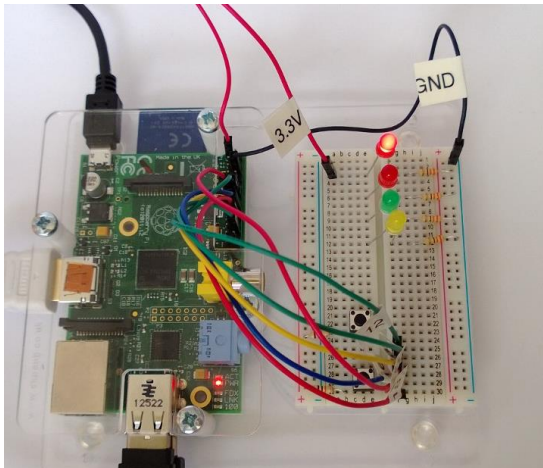


Figure 11: Testing a LED

You are going to use the 3.3volt cable and check each LED in turn. By plugging the 3.3 volt plug into each row in turn you can make sure that all the LEDs work correctly. As supplied our board should have all the components fitted, but the wires for the GPIO pins should be “parked” in an area of the breadboard that we are not using at the moment. There are also some buttons on the board; you can ignore these for the moment.



Test each LED in turn to make sure that it lights up. Make sure that you put the power pin into the correct column that lines up with the LED. If the LED doesn't light up this may be because it plugged in the wrong way round. The long leg of the LED should be in the hole nearest to the Raspberry Pi board. If you have checked this and the light still doesn't come on, make sure that your Raspberry Pi board is powered on, and that the GND cable is connected correctly to the bottom row.

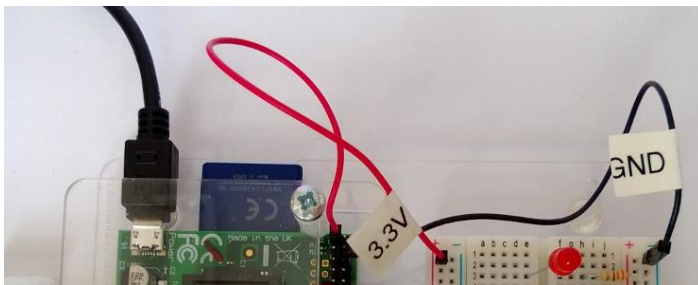


Figure 12: Final Settings

When you have finished you can connect the 3.3V supply to the top rail so that it can be used to power hardware that we will use later in the lab.

Once you have lit some LEDs you are ready to move onto the next stage of the work, which is where we will start controlling the lights using software.

## Controlling Outputs with Software

At the moment we have just connected our LED to the Raspberry Pi power supply. This is fine if we just want a light, or something that indicates that our system is switched on, but is not very interesting. To get started we need to understand how software works on the Raspberry Pi and how we can write code that controls the LEDs. We are going to use a language called Python.

### Starting up the Raspberry Pi

We are going to write our programs using a programming environment that works within the X-Windows environment on the Raspberry Pi. This might not be the environment that you have with your device, but the systems you are going to use here have been configured to work this way.

### Logging On

You might not have to do this, but if you do – here's how.

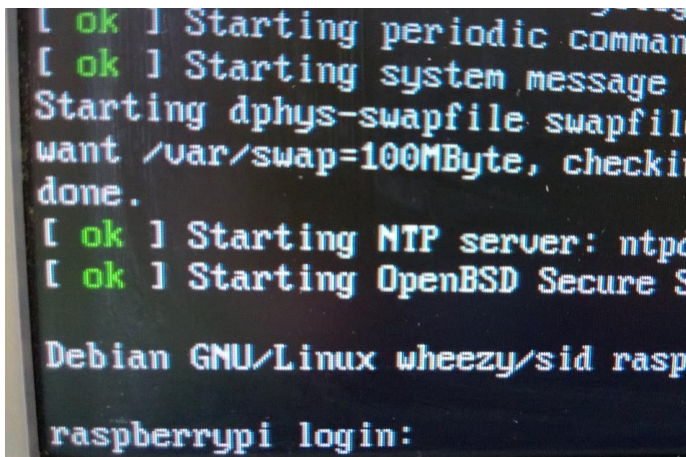


Figure 13: Raspberry Pi login screen

When the Raspberry Pi starts running from power up the first thing you need to do is log on to the system. Figure 13 shows the login prompt for the system. Having to log in like this is nothing new; pretty much every system that you use today will require a user to authenticate before they can use it. In the case of the Raspberry Pi a brand new system will have a user called **pi** with a password which is **raspberrypi**.

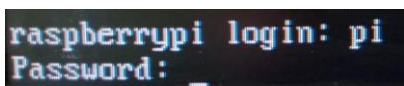


Figure 14: Typing in the username

Type in the username “pi” and then press enter, as you can see in Figure 14. The system will ask for the password. Type in “raspberrypi” (without the quotes and in lower case letters) and press enter again.

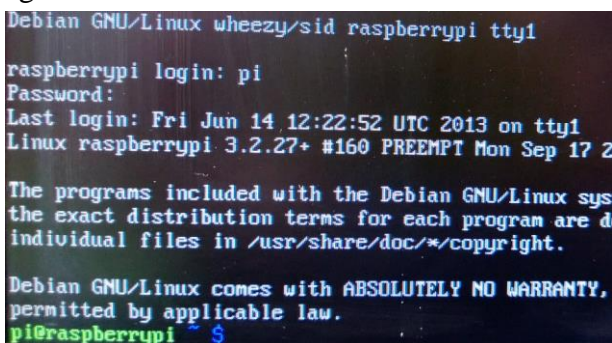


Figure 15: A successful login

If the username and password match you will be logged in, and see the display in Figure 15.



If you want to you can change the password to something else. If you do this, make sure you remember what the new password is, otherwise you may be locked out of your computer for ever. To keep things simple, I'm not going to tell you how to change the password, and I'd rather you didn't do this to the machine we will be using in the lab.

Once you have logged in you have a *terminal window* which you can use to give commands to the Raspberry Pi operating system. There are commands to start and stop programs, create and delete files and do lots of other things to the computer.

## Shutting down a Raspberry Pi

It is not sensible to just turn off a Raspberry Pi. This might leave parts of the operating system in a state which means that the device will not wake up again. This is not a huge problem because you can always wipe the memory card that holds the Raspberry Pi operating system and copy a replacement onto it but if this means wiping some of your precious data it might be a bit tiresome. The command “halt” will shut your system down, but this is a command that cannot be used by mere mortals. Only super-users can shut the system down. Fortunately the pi user (the one you logged in with) has the ability to run commands in the role of a super user. To run any command as the super user you precede the command with `sudo` (short for “super user do”). So, to shut down your Raspberry Pi you give the command:

```
sudo halt
```

The system will display a bunch of messages as the operating system shuts down. Once these have finished the screen will go blank and you can turn off the power. Any files that you have created and saved on files in the operating system will be there when the system restarts.



Shutting down and restarting a system takes a few minutes, so only shut it down if you are sure you won't need it for a while.

## Starting X Windows

Once we are logged in we can start the windowed environment that we will be using to create our code. This is slightly complicated because the programs we are writing are going to talk to the hardware. This is something which is not normally permitted. Only super users can talk to the hardware, because a badly behaved program could twiddle with hardware elements and potentially break the system. We are not going to do anything like this ourselves, we are just going to use the Python methods that let our programs talk to hardware, but nevertheless we need to start X windows (the place where we are going to write the software) as a super user.

```
sudo startx
```

If you give this command the system will start up a windowed environment that looks a lot like ones you have seen before.



If you have used the X-Windows environment before on your Raspberry Pi you might have had some shortcuts on the desktop background. These provide quick access to some programs. When you run X-Windows as super user the system does not give you these shortcuts. This is not actually a problem, as we can run the programs from the start button anyway.

Once you have X-Windows running you can then open up the Python environment from within it.



Get X-Windows running on your system.



## Using the IDLE 3 environment to write Python programs

We are going to use a Python environment called IDLE. This lets you create and run Python programs.

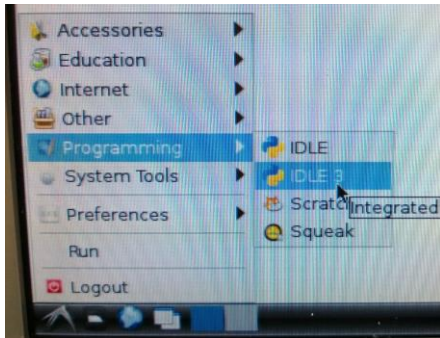


Figure 16: Starting Idle 3

To start IDLE 3 you should use the mouse to click on what I'm probably not allowed to refer to as the Start Button in the bottom left hand corner of the screen. Then select the Programming item in the menu that appears and then the IDLE 3 item from the sub menu.



Figure 17: The Busy indicator

The busy indicator shows how much work the Raspberry Pi is doing. The more green that you can see, the harder it is working. Figure 17 shows what happened when I started up IDLE3. If you start something and nothing much seems to be happening it is worth checking this indicator to see if the Raspberry Pi is busy. If the indicator is fully green it might be worth waiting for the Raspberry Pi to catch up.

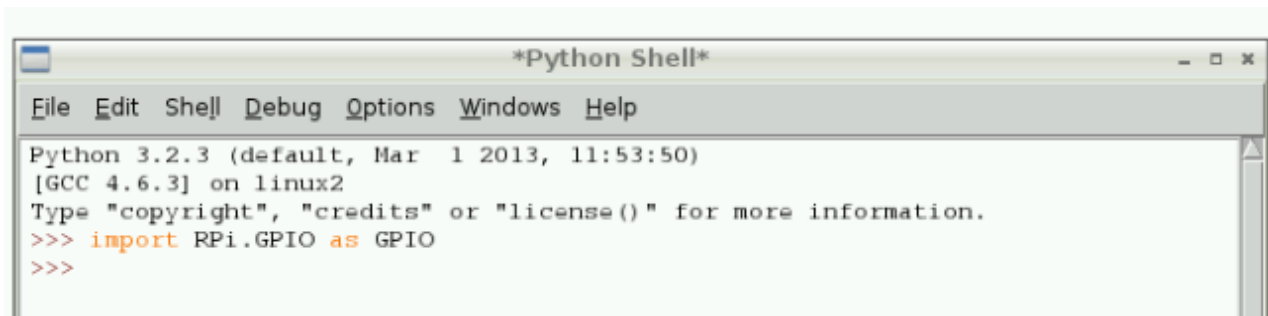
Once you have IDLE3 running you can start to give it commands. The environment will take each Python statement and obey it. Type the command and press the Enter key.

### Importing the GPIO Library

Python has a bunch of commands that can be used to drive the hardware pins. Before we can actually use these commands to drive the hardware we need to import them into the program. This is a bit like fetching that special tool from the shed that we use to do a particular job. The Python statement that does this is the import statement

```
import RPi.GPIO as GPIO
```

This identifies the library, imports it and gives it a name that we will use to refer to it in our program. I've given it the internal name GPIO. We could use any name here, for example CHEESE or GARY, but this would make the program confusing. From now on, whenever we say GPIO.something the Python system will look in the GPIO library to find that something. When you get heavily into Python you will find that there are lots of libraries like this which you can use in your program, in fact we will use another library later on.



```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.3 (default, Mar 1 2013, 11:53:50)
[GCC 4.6.3] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> import RPi.GPIO as GPIO
>>>
```

Figure 18: Typing a command in to the Python shell

Figure 18 shows how you enter commands to the Python Shell. You may need to click on the shell window before you can type text into it.



Type the statement above to import the GPIO library into IDLE3. If you get this right you should see nothing happen. The >>> prompt will just come back. This is good news of a sort I guess. Remember to press Enter when you have typed the command.

## Configuring the Pin Numbers on the hardware

The next thing we need to do is tell the Raspberry Pi that we are using the pin numbers on the Raspberry Pi circuit board to identify which signals we want to work with.

You can also refer to these signals by their port numbers on the internal chip that actually makes them work. This can be useful if you are a highly technical sort, for our purposes it is easiest if we just use numbers that match the pins on the board. As we shall see in a moment, we are going to use a cunning plan to make it very easy for us to change which connections are wired to which pin. To get the GPIO library to do things for us we will call methods in the library. A method is a lump of program code that has been given a name. We might write our own methods later on, for now we are just going to call methods that other people have written. The method in the GPIO library that we use is `setmode`:

```
GPIO.setmode(GPIO.BOARD)
```

This method is told the numbering mode to use in this case it is the value `GPIO.BOARD`.



Type the statement above to set the mode for the input pins.

If the command doesn't work, make sure that you have got the capitals and little letters exactly right. You should also make sure that you aren't replacing letters with numbers, for example it is "GPIO" (rhymes with "Yippie'Eye'O") and not GP10 (which is "GP ten").

If you use the wrong command, for example type `SetMode` with some capital letters, then you will get a message from Python saying that it doesn't know how to do that task. It will look something like this:

```
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    GPIO.SetMode(GPIO.BOARD)
AttributeError: 'module' object has no attribute 'SetMode'
```

This is how Python says it doesn't understand what you have just typed. Don't worry if you see messages like this, you haven't done anything that will damage the program. Just re-type the command correctly and all will be well.

## Setting up the Pins

So far we have managed to get the Python environment running, the hardware libraries installed and set up so that the numbers of our pins match the ones on the hardware. Now we want to write some code that will control one of our leds.



First select the cable tagged with 11. The number on the tag matches the pin that it is connected to on the Raspberry Pi. Plug it into the breadboard so that it is connected to the right hand LED.

We don't want to have to remember the number 11 all the way through our program, so instead I'm going to create a variable with the value 11 in it.

```
right_red_led = 11
```

When Python sees this it creates a variable called `right_red_led` and puts the value 11 into it. You can think of a variable as just a named box which can hold a number.



Type in the statement above to create a `right_red_led` variable set to 11. Now make sure that the variable has the right number in it by printing it out.

If you want to find out what is in a variable you can just type the name of the variable, and Python will print the value for you.

```
right_red_led
```



Type in the statement above to make sure that the variable that you have created has the number in it that you expected.

A program can contain many variables. The Python will keep track of where they are stored, you just have to think of names for each one.



If you miss-type the name of a variable this will cause problems that can sometimes be hard to track down. If I type `Right_Red_Led` at some later point in the code the Python system might think I'm referring to a brand new variable at that point, with subsequent "hilarious" results. If you find your program doing silly things (and this does happen – even to the best of us) make sure that the names of all your variables have been typed correctly.

Next we need to configure the pin as an output. The `GPIO.setup` method will do this for us:

```
GPIO.setup(right_red_led,GPIO.OUT)
```

If we want to configure the pin as an input (we'll do this later) we'll use the setting `GPIO.IN`.



Set the red led up as an input using the statement above.

At this point the output might not be doing much. Normally when a new output port is created it is set to low, i.e. no voltage. This means that the LED will not be lit. However, we can use some code to turn it on.

```
GPIO.output(right_red_led,True)
```

The `GPIO.output` method needs to be told two things; the thing that is being controlled and the value to set it.

The first item given to the method is the number of the LED, which we pass via the variable that we created earlier. Note that I could have put 11 here (if I'd wired the LED to pin 11) but this would be much harder to understand.

The second item is a boolean value. Boolean logic is a way that a program can represent things that only have to possible states. In the case of Python these two states are given the name `True` and `False`. In the case of the `GPIO.output` method the value `True` means output 3.3 volts and the value `False` means output 0 volts. In a Python program we can use the values `True` and `False` as *literal* values (which are "literally just there"). So we can represent numeric values by writing values such as 11, and we can represent Boolean values by using `True` and `False`.



Use the statement above to call the output method to turn the right red LED on. Note that the LED should now light. If it doesn't you might want to check the wiring. Once you have turned the light on, prove that this was not a fluke by turning the light off again.

## Flashing the LED

Turning the light on and off is all very well, but at the moment it is a bit boring. Let's try to make the light flash. To start with we will make the light flash forever, and we can use a Python looping construction to do this for us. The while construction lets us repeat a set of statements while a test is true.

```
while True:
    GPIO.output(right_red_led,True)
    GPIO.output(right_red_led,False)
```

The statements above will repeatedly turn the LED on and off for ever. (don't worry, we can stop them).

The way that Python works, all the statements that are indented to the right will be repeated by the loop. When the end of the indented statements is reached, Python will go back to the while line, test the condition and then, if the condition is True, go round again. Since our condition is always true, this means that the loop will run for ever.



Type in the statements above. Be careful to exactly match the code as written. Don't forget the : (colon) at the end of the while line and make sure that True and False have capital letters.

You'll notice that after you have typed the while statement the editor will automatically indent when you press Enter so that you can type the GPIO.output statements. When you have typed the last line, press Enter twice in succession to finish entering the loop and start it running. However, you might not see that the light is flashing.

The problem is that the Raspberry Pi is quite a powerful little computer. It can run many thousands of lines of Python a second. The LED is being turned off and on so quickly that you can't see the flashes. While your program is running, take a look at the performance display at the bottom right of the window. You should see that it is fully green, which means that our little program is using up just about the entire computer.

To stop the program you just have to hold down the CTRL key and press C. This will interrupt the program and you should see the performance indicator drop back to next to nothing.



Press CTRL+C and stop the program.

## Taking control of Time

We need something that we can use to pause the program so that we can see the light flash. Fortunately Python provides a set of methods that will do this for us. These are in the time library, so we have to import this into our code.

```
import time
```

I love this statement. I keep expecting a Tardis to appear each time I type it in. However, all it does is provide access to some methods that I can use to pause the program (A while back we did something similar to get hold of the GPIO libraries). The method we are going to use is sleep.

```
time.sleep(3)
```

The above method would pause the program for three seconds.





Use the statement above to import the time library and issue the command

```
time.sleep(3)
```

You should notice that it takes three seconds for the command prompt (the >>> sequence) to reappear.

We can see our LED flash by making a loop that pauses each time after the LED has been turned on or off:

```
while True:
    GPIO.output(right_red_led,True)
    time.sleep(0.5)
    GPIO.output(right_red_led,False)
    time.sleep(0.5)
```



Type in the above code and your LED should flash. The delay is half a second, so the light should flash once a second. Try using different values to make the LED flash more quickly, or pulse (on for less time than it is off).

## ***Controlling multiple outputs***

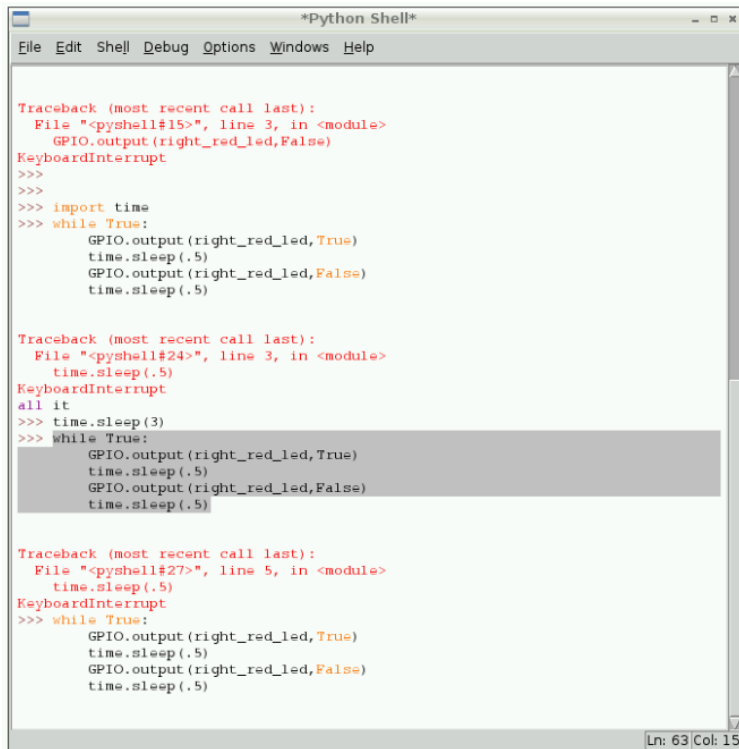
Now that you have made one light flash you can move on to controlling two. See if you can make the kind of flashing behaviour that you see in rail crossing lights, where the red LEDs flash alternately. You will need to wire up another LED and configure it as an output to do this. You can use any of the numbered connectors to control the other LED, simply create a variable (perhaps called `left_red_led`) and set it to indicate which connection you are working with. There are also yellow and green LEDs, so you can work up a complete traffic light display if you really fancy it.

## **Working with Python code**

At the moment we have been using the Python Shell part of IDLE to enter Python program statements which are obeyed instantly. This is a great way to experiment with the language, but we have discovered that if we want to run the program more than once we have to type the commands again. You might be getting tired of typing the same things time and time again. In this section we are going to find out how you can create and save your program files and also how you can use cut and paste to reuse your “experimental” program code.

### ***Using Copy and Paste***

Inside IDLE you can use copy and paste to move text from one part of the program to another. You can use the mouse to select a block of text that you have already entered and then you can copy the text by holding down the CTRL key and pressing C. Then you can click on the cursor at the destination position and press CTRL+V to paste that text. (When I write CTRL+key it means hold down the CTRL key and press the one indicated).



```
*Python Shell*
File Edit Shell Debug Options Windows Help

Traceback (most recent call last):
  File "<pyshell#15>", line 3, in <module>
    GPIO.output(right_red_led,False)
KeyboardInterrupt
>>>
>>>
>>> import time
>>> while True:
    GPIO.output(right_red_led,True)
    time.sleep(.5)
    GPIO.output(right_red_led,False)
    time.sleep(.5)

Traceback (most recent call last):
  File "<pyshell#24>", line 3, in <module>
    time.sleep(.5)
KeyboardInterrupt
all it
>>> time.sleep(3)
>>> while True:
    GPIO.output(right_red_led,True)
    time.sleep(.5)
    GPIO.output(right_red_led,False)
    time.sleep(.5)

Traceback (most recent call last):
  File "<pyshell#27>", line 5, in <module>
    time.sleep(.5)
KeyboardInterrupt
>>> while True:
    GPIO.output(right_red_led,True)
    time.sleep(.5)
    GPIO.output(right_red_led,False)
    time.sleep(.5)

Ln: 63 Col: 15
```

Figure 19: Selecting Text

IDLE will show you which text has been selected by highlighting it in grey, as shown in Figure 18. When you press CTRL+C the text that you have marked is copied and you can move to the place you want to insert the text and press CTRL+V to paste it.

When you insert the text it will be used by IDLE just as if you had typed it. You can select lots of lines of text if you want to repeat the whole thing. You might need to press Enter at the end of the last line to input that.



As a practice, you might like to stop your flashing light program and then cut and paste the entire while construction to run it again.

If you don't want to use the mouse to select text you can do this by moving the cursor around using the arrow keys and then holding down the SHIFT key and moving the cursor to select blocks of text. If you want to save a particularly useful lump of Python for later you can use the File menu in IDLE to save your session into a text file. You can open the file later and use the code from it.

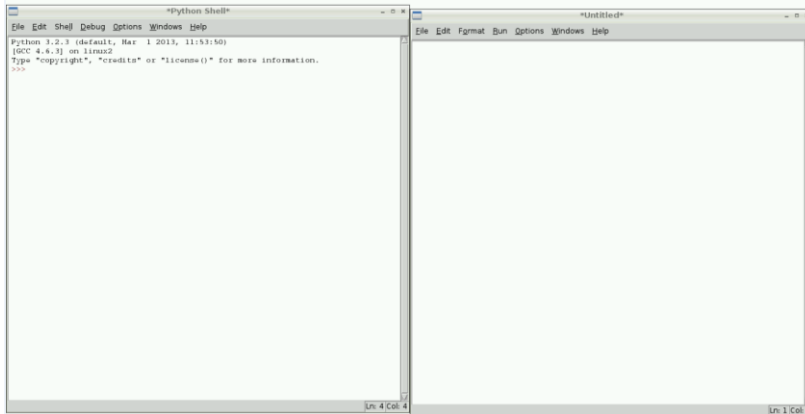
## Creating Complete Python Programs

At the moment we have been using the Python Shell part of IDLE to enter Python program commands. This is a great way to experiment with the language, but we have discovered that if we want to run the program more than once we have to type the commands again. We can do clever things with copy and paste (see above) but what we really want to do is store our programs and load them again. It turns out that we can do this in IDLE quite easily.

From within IDLE we can open up a new window where we can work on Python programs. When we think the program might work we can then ask IDLE to run the program so that we can see whether or not the program works properly. This is exactly what professional developers do when they write programs.

To create a Python program the first thing you need to do is open a new Window on the desktop. Click on the File tab on the top line of the IDLE window and select New Window from the menu that appears. Alternatively, you can press CTRL+N.

This should cause a second window to appear on the screen, which will have the helpful title *\*Untitled\**.



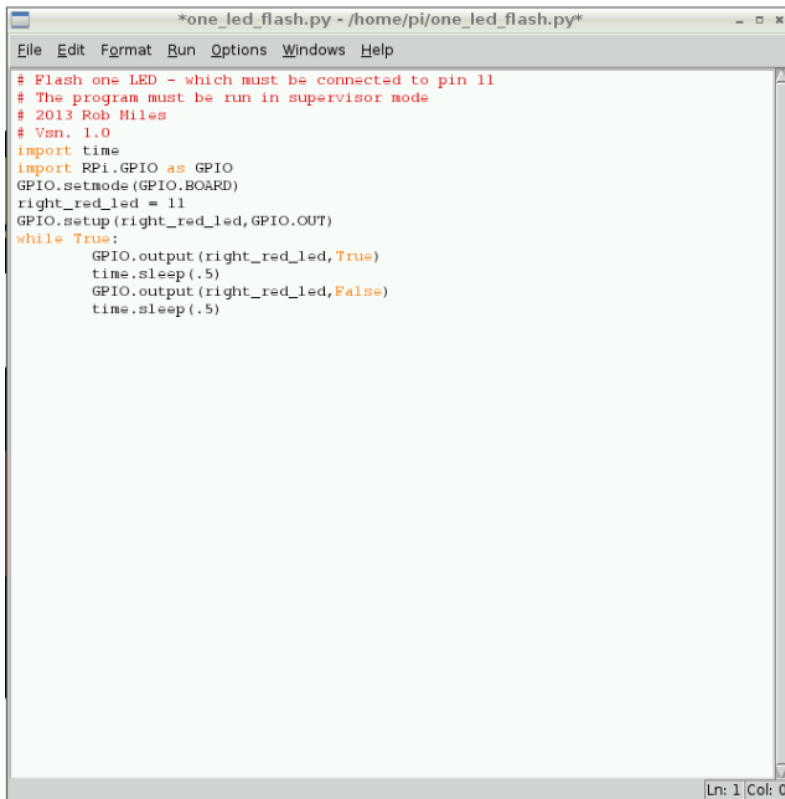
*Figure 20: Program window and Python Shell*

Figure 20 above shows the two windows at the start of a Python development session. Unlike the Python shell window, the program lacks the `>>>` prompt where you type commands. You can type lines of Python into the new window and they are not executed. They are held as a program.

You can treat this window as any other text editor that you have used before. Think of it as a word processor for programs. You can type in the statements that make up the single LED flashing program as below. Alternatively you can use the cut and paste features we saw previously to get the text into the window. Below you can see a complete Python program that will flash a LED connected to pin 11.

```
# Flash one LED - which must be connected to pin 11
# The program must be run in supervisor mode
# 2013 Rob Miles
# Vsn. 1.0
import time
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BOARD)
right_red_led = 11
GPIO.setup(right_red_led,GPIO.OUT)
while True:
    GPIO.output(right_red_led,True)
    time.sleep(.5)
    GPIO.output(right_red_led,False)
    time.sleep(.5)
```

Note that I've added some lines that start with the hash (#) character. These are comments. The Python system will completely ignore lines that start in this way. I use comments to leave notes for myself about what the program does and any other things it might be useful to know about how it works.



```
*one_led_flash.py - /home/pi/one_led_flash.py*
File Edit Format Run Options Windows Help
# Flash one LED - which must be connected to pin 11
# The program must be run in supervisor mode
# 2013 Rob Miles
# Ver. 1.0
import time
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BOARD)
right_red_led = 11
GPIO.setup(right_red_led, GPIO.OUT)
while True:
    GPIO.output(right_red_led, True)
    time.sleep(.5)
    GPIO.output(right_red_led, False)
    time.sleep(.5)
Ln: 1 Col: 0
```

Figure 21: A program entered into the program window

Figure 21 shows the program as it appears after being typed in. Note that the editor “colours in” some of the words that are part of the Python language, for example `while` and `True` are shown in gold. I’ve saved this program in a file called `one_led_flash.py` and so the title of the window has been changed to reflect this. I used the `Save` command on the `File` menu to save the program text in a file.

You can now create Python programs and save them to work on next time. You can also email your Python code to other people to have a look at and work with and you can run programs that they send you. If you use the `Open` command from the `File` menu in IDLE3 you can open a file that you have already created and work on it later.

## Running Python Programs

Once you have entered the program code you can run it by opening the `Run` menu from the top line and selecting `Run Module` from the menu. Alternatively you can press the `F5` function key on your keyboard. The first time you run a new program the system will insist that you save the program in a file somewhere.

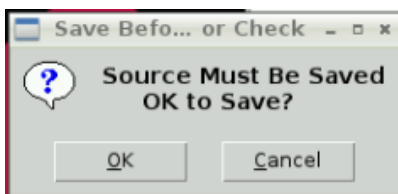
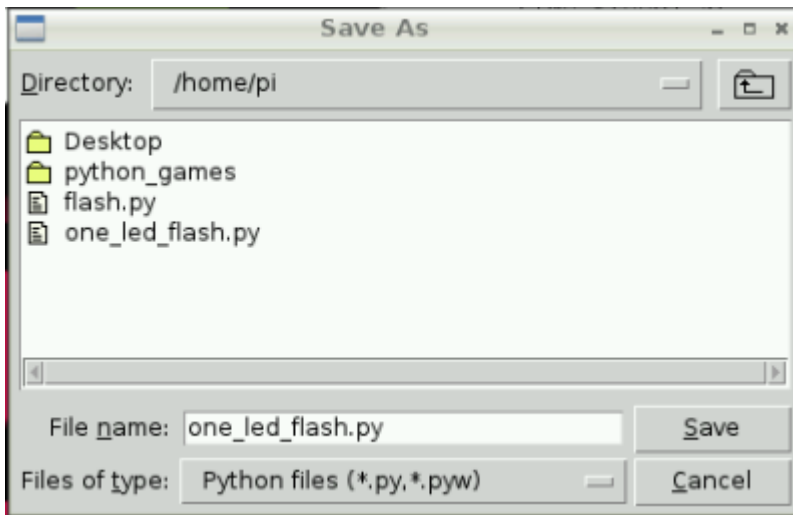


Figure 22: Source save request

Figure 22 shows you the save request. You will see this each time you try to run the program. You **must** save the file in order to be able to run it. Choose a name for the file and then save the program in that file. Make sure that the name you use has the characters `.py` at the end so that the Raspberry Pi operating system knows the file contains a Python program.



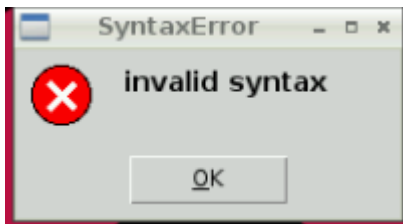


*Figure 23: Save file dialog display*

The save commands are very similar to the ones that you have seen on other computer systems. Figure 23 shows you what they look like. You can move around the file store and pick somewhere to put the files. Note that in the above dialog I have already saved a couple of Python programs.

### **Program syntax errors**

When you start the program running the IDLE environment checks your program to make sure it is correct and then runs it in the IDLE Python shell. If you have made any mistakes you will see these described in the shell and you will have to look in your edit window to see where the mistake is.



*Figure 24: The dreaded syntax error display*

If the Python runtime system doesn't like your program it will display the error box shown in Figure 24 above. This means that you must look through your program to find out where you went wrong. There are lots of reasons why Python might not like your program. You might have misspelled a name, for example typed `true` rather than `True`, or you might have missed out an important character. Another reason why you might get problems is that Python uses the layout to work out how the program is to be run. If you indent a line that is not supposed to be indented that will cause an error too.

```
reaction_game.py - /home/pi/reaction_game.py
File Edit Format Run Options Windows Help
yellow_led = 16
green_led = 15
left_switch = 22
right_switch = 18
# Set up the ports
GPIO.setup(right_red_led, GPIO.OUT)
GPIO.setup(left_red_led, GPIO.OUT)
GPIO.setup(yellow_led, GPIO.OUT)
GPIO.setup(green_led, GPIO.OUT)
GPIO.setup(left_switch, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
GPIO.setup(right_switch, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
# Turn off the yellow LED and the red LEDs
GPIO.output(yellow_led, False)
GPIO.output(left_red_led, False)
GPIO.output(right_red_led, False)
# Wait for both buttons to be pressed
while True:
    if (GPIO.input(left_switch) == True and GPIO.input(right_switch) == True):
        break;
        time.sleep(.01)
# When we get here both buttons have been pressed
# Turn on the yellow LED
GPIO.output(yellow_led, True)
# Wait five seconds
time.sleep(5)
# Turn on the green LED
GPIO.output(green_led, True)
# Wait for a button to be released
while True:
    if (GPIO.input(left_switch) == False or GPIO.input(right_switch) == False):
        break;
        time.sleep(.01)
# When we get here one of the buttons has been released
# Show the winner
if (GPIO.input(left_switch) == False):
    GPIO.output(left_red_led, True)
if (GPIO.input(right_switch) == False):
    GPIO.output(right_red_led, True)
```

Figure 25: A broken program

Figure 25 shows how Python tells you about errors. This is quite a complicated program which we'll be looking at later, but I've missed a colon (:) off the end of the line near the bottom. Python is rather helpfully showing me the fault with a big red bar.

The Python system will do its best to show you where you have gone wrong, but it is not perfect. It will show you the place in the program where it noticed the error, not where you actually made the mistake. This means that sometimes the red bar will not be on the error, but somewhere else. Finding syntax errors like this is a bit of an art, and something that you will get better at with practice.



Create a Python program (use the one I wrote above) and save it. Load the file and run it from within IDLE.

## Reading Inputs

At the moment we can just drive outputs. This is nice enough, but we really need to be able to read inputs as well. This is easy enough to do, let's start by creating an input port.

```
left_switch = 22
```

I'm going to use pin 22 as an input, you can use another pin if you like. Plug the cable for the pin that you have selected into the top rail on the breadboard, so that it is connected to the 3.3 volt power supply from the Raspberry Pi.

Now we need to tell the GPIO library to use this pin as an input:

```
GPIO.setup(left_switch,GPIO.IN)
```

Once we have our input we can now read from it.

```
while True:  
    print(GPIO.input(left_switch))
```

The above lump of Python repeatedly reads from the input switch and prints out the value it gets. This is the complete program that sets up our input and reads it:

```
left_switch = 22  
GPIO.setup(left_switch,GPIO.IN)  
while True:  
    print(GPIO.input(left_switch))
```



Create a new program called `input_test.py`

Type in the code and start it running. You should see a whole bunch of the value 1 being printed out. This is because pin 22 (or whichever pin you have used) is connected to the 3.3 volt supply.

While the program is running, unplug your cable and watch what happens when you hold the pin in the air. You will see a succession of 0 and 1 values displayed. If you now plug the cable into the bottom row and connect it to the GND signal (0 volts) you should see that the value turns to 0.

The random 1s and 0s that you saw are caused by signals that are induced in the cable by electrical interference in the cable, the same kind of thing that causes old radios and TVs to hiss and crackle. The problem is caused because with nothing connected to it the cable functions a bit like a tiny aerial and picks up any voltage from surrounding devices, including mobile phones, mains cables and even people.

We can stop this noise from being induced into the input by configuring the GPIO port to have a tiny resistor connected to it which will pull the signal one way or another (up or down) when there is nothing connected to it. To do this we just have to add more instructions to the GPIO port when we create the input:

```
GPIO.setup(leftSwitch,GPIO.IN,pull_up_down=GPIO.PUD_DOWN)
```

The extra information tells the setup method that the port should always be pulled down when it is not connected to anything else.



Change your program to use the new setup above.

Re-run your program with the modified setup method. Remember that you can stop your program which is displaying the output by pressing CTRL+C.

You should find that with the pull up enabled the line doesn't print noisy results any more, it just shows a constant 0 when it is not connected. This shows that the pull down resistor is pulling the input signal down to 0 for us.

The breadboard has a tiny button on it which is connected via a resistor to the 3.3 volt rail. You can connect your input signal to the output of the button so that when the button is pressed the input is made high, and the port returns the value 1. We can then use this to read input from the user of our device.

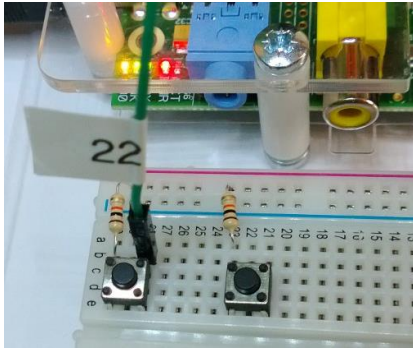


Figure 25: Connecting to an input button.

Figure 25 shows how the button is fitted; when the button is pressed it connects the resistor to the input cable. The resistor is there just in case the other side of the button gets connected to the ground level. If this happens the resistor will stop too much current flowing and damaging the Raspberry Pi power supply. There are two buttons available. To connect the second button you need to plug a signal cable into the socket that lines up with the output from that button.



The hardware on the breadboard is pulling the input high when the user presses the button, and we have a pulling resistor which pulls the signal low when nothing is connected to it. This makes perfect sense if you think about it. And while you are thinking about it, you might want to consider what changes we would have to make if we had a button that connected the input to GND rather than 3.3 volts.

If you connect your input cable to the button you should find that the test program will print 1 when the button is pressed and 0 when the button is released.

## Creating a Reaction Timer Game

Now that we have our inputs and outputs we can create a simple reaction timer game. The game will test who can release their button the fastest when they see the trigger led. It can get quite competitive. The game is for two people and will work like this:

1. Load the libraries
2. Select the pin numbers
3. Setup the hardware
4. Turn off all the LEDs.
5. Wait for the players to press their buttons and hold them down.
6. Turn on the yellow LED to indicate that the game is starting.
7. Wait five seconds.
8. Turn on the green LED.
9. Wait until one of the players releases their button.
10. The first player to release their button is the winner. Light their LED to indicate this.

We are going to write this as a Python program, learning how Python works as we go. We will consider how to make each of the ten steps in turn.

Some of the steps need Python constructions that we haven't seen before. Don't worry about those though; just think in terms of what the program needs to do at that point to make the game work.



## 1. Load the libraries

```
# Hardware Reaction Game
# Uses four outputs (two red and one yellow and one green)
# and two switch inputs
# The program must be run in supervisor mode
# 2013 Rob Miles
# Vsn. 1.0
# imports
import time
import RPi.GPIO as GPIO
```

We have seen this code before. We have a helpful set of comments that act as a heading for the program and then we import the two Python libraries that we have seen before.



1. Open a new program window and type in the code.
2. Save the program in a file called `reaction_timer.py`
3. You can run the program if you wish. It should not do anything.

Now that we have the Python environment all set up we can select the pins we are going to use.

## 2. Select the pin numbers

```
# Configure the hardware to use Raspberry Pi input ports
GPIO.setmode(GPIO.BOARD)
# Pin assignments for the LEDs
right_red_led = 11
left_red_led = 12
yellow_led = 16
green_led = 15
left_switch = 22
right_switch = 18
```

This is the code that selects which connections we are going to use for each of the signals.



1. Add the statements above to the end of your program.
2. You can run the program again if you wish. It should still not do anything.

You can use your own numbers for the pin assignments if you like, but I think it is easier if you use the ones above. A benefit of this way of working is that if we do have to change the pins at a later time we just have to change these lines to make our program work with the new hardware. These statements also make it very clear which pin is doing what, which is a good thing too.



The numbers must match the ones that you connect on the breadboard. Otherwise you will not get the gameplay experience that you are expecting, and you might damage the Raspberry Pi if you apply a signal from the button into an output the Raspberry Pi is trying to control.

### 3. Setup the hardware

```
# Set up the ports
GPIO.setup(right_red_led,GPIO.OUT)
GPIO.setup(left_red_led,GPIO.OUT)
GPIO.setup(yellow_led,GPIO.OUT)
GPIO.setup(green_led,GPIO.OUT)
GPIO.setup(left_switch,GPIO.IN,pull_up_down=GPIO.PUD_DOWN)
GPIO.setup(right_switch,GPIO.IN,pull_up_down=GPIO.PUD_DOWN)
```

You have seen all these statements before. There is one statement for each pin we are using. Note that we are using pull down resistors on the inputs.



1. Add the statements above to the end of your program.

### 4. Turn off all the LEDs

```
# Turn off the LEDs
GPIO.output(yellow_led,False)
GPIO.output(green_led,False)
GPIO.output(left_red_led,False)
GPIO.output(right_red_led,False)
```

When the game starts the program must turn off all the leds. These statements do just that.



1. Add the statements above to the end of your program.
2. You can run the program. It will turn the LEDs off, but as they were probably off before you might not notice.

### 5. Wait for the players to press their buttons and hold them down

```
# Wait for both buttons to be pressed
while True:
    if (GPIO.input(left_switch)==True and
        GPIO.input(right_switch)==True):
        break
    time.sleep(.01)
```

The next part of the program is full of new stuff. However, it is not too hard to understand if you focus on what we are trying to achieve. We want the program to wait until both buttons have been pressed. If you were working as the computer and you were asked to wait until you saw both buttons pressed you would repeatedly check until you saw both buttons pressed, and then you would move on to the next step. That is exactly what this code does.

We know that we can make a program loop using a while construction. Python can also do tests using an if construction. The if construction is followed by a condition which can be True or False. We can also combine conditions by using the and operator. Finally, we can break out of a loop by using the break statement.

The program repeatedly tests the buttons and breaks out of the loop when both buttons are pressed. The time.sleep statement is obeyed each time round the loop. It is just there so that our program doesn't take up the entire Raspberry Pi when it runs. A delay of 100<sup>th</sup> of a second each time round will not affect the performance performance of the game (unless Superman is taking part) and it will give other systems on the Raspberry Pi a chance to run.



1. Add the statements above to the end of your program.
2. You can run the program. This time it will do something. The IDLE Python shell will pause until you press both of the input buttons.

If this code seems hard to understand (and it is a bit tricky the first time) remember what you are trying to do. You want the program to wait until both players have pressed their buttons and the game is ready to start. The inputs from the buttons are high (True) when a button is pressed and you want to make sure that the left switch **and** the right switch are pressed at the same time.



If you are feeling clever, you can use the fact that the input methods return True or False to make the above program a bit smaller. But you don't have to do this.

## 6. Turn on the yellow LED to indicate that the game is starting

```
# When we get here both buttons have been pressed
# Turn on the yellow LED
GPIO.output(yellow_led, True)
```

This is easy code. It just turns on the yellow LED. This tells the players that the game is ready to start.



1. Add the statements above to the end of your program.
2. You can run the program. This time it will do something too. When you press both buttons the yellow LED will light.

## 7. Wait five seconds

```
# Wait five seconds
time.sleep(5)
```

Another simple statement. This just waits five seconds. The program will pause at this point. We could make the program more interesting by pausing for a random number of seconds rather than the same amount each time. Python provides libraries that will produce random numbers.



1. Add the statements above to the end of your program.
2. You can run the program. This time it will do something too. When you press both buttons the yellow LED will light and the prompt will take five seconds to return.

## 8. Turn on the Green LED

```
# Turn on the green LED
GPIO.output(green_led, True)
```

This statement is simple too. It just turns on the green LED. This is the signal for the players to release their buttons. The first player to release their button wins the game



1. Add the statements above to the end of your program.
2. You can run the program. This time it will do something too. When you press both buttons the yellow LED will light and then, after five seconds the green LED will come on.

## 9. Wait until one of the players releases their button

```
# Wait for a button to be released
while True:
    if (GPIO.input(left_switch)==False or GPIO.input(right_switch)==False):
        break
    time.sleep(.01)
```

This is another rather complicated piece of code. It is very similar to the loop at step 5, but rather than waiting until both buttons have been pressed down, (`left_switch` and `right_switch` both `True`) it now waits until one of the buttons is released (`left_switch` or `right_switch` now `False`). As before, the program breaks out of the loop when a button is released, and as before, it delays for a hundredth of a second each time round to avoid stealing the entire computer.

Note how the indenting (the way the program text is moved in from the left margin) shows how each statement is controlled by the one above it. The `if` and the `time` statements are both controlled by the loop, but the `break` statement is controlled by the loop and the `if` condition. Should we want to do lots of statements when the button is released (perhaps we might want to play a sound) then we would indent them at the same level as the `break`. Of course we'd have to put those statements before the `break` statement because once the `break` is obeyed the program leaves the loop completely and continues at the statement that immediately follows the loop.



1. Add the statements above to the end of your program.
2. You can run the program. Now you can go through the complete game sequence. The yellow LED will come on until you hold down both buttons. Then, after five seconds the green LED will come on. The program will remain running until you release one of the two buttons, at which point the IDLE prompt will reappear.

## 10 Light the winning LED

```
# When we get here one of the buttons has been released
# Show the winner
if (GPIO.input(left_switch) == False):
    GPIO.output(left_red_led, True)
if (GPIO.input(right_switch) == False):
    GPIO.output(right_red_led, True)
```

When the program breaks out of the loop we know that one of the players has won, but we don't know which one. The statements above just test each switch in turn and light the LED that matches the switch that has been released. Note that we have to test to see if the input is false for a switch, because that is how we know that the switch has been released.



1. Add the statements above to the end of your program.
2. You can run the program and play the game all the way to the end.

If the program doesn't do what you are expecting, make sure that all the connections are correct and that the numbers on the pins match those in the program. Make sure that the statements are indented correctly. If you get the sequence or the tests the wrong way round you get a *run time error* which means that the program runs but doesn't do precisely what you want. Fixing these is the mark of a really skilled programmer.



There is actually quite a serious bug in this program that makes it very easy for a player to cheat. See if you can figure out what you could do so that you are guaranteed to win every time, and how you would extend the program to stop this kind of cheating.

## Complete Reaction Timer Program

```
# Hardware Reaction Game
# Uses four outputs (two red and one yellow and one green) and two switch inputs
# The program must be run in supervisor mode
# 2013 Rob Miles
# Vsn. 1.0
# imports
import time
import RPi.GPIO as GPIO
# Configure the hardware to use Raspberry Pi input ports
GPIO.setmode(GPIO.BOARD)
# Pin assignments for the LEDs
right_red_led = 11
left_red_led = 12
yellow_led = 16
green_led = 15
left_switch = 22
right_switch = 18
# Set up the ports
GPIO.setup(right_red_led,GPIO.OUT)
GPIO.setup(left_red_led,GPIO.OUT)
GPIO.setup(yellow_led,GPIO.OUT)
GPIO.setup(green_led,GPIO.OUT)
GPIO.setup(left_switch,GPIO.IN,pull_up_down=GPIO.PUD_DOWN)
GPIO.setup(right_switch,GPIO.IN,pull_up_down=GPIO.PUD_DOWN)
# Turn off the LEDs
GPIO.output(yellow_led,False)
GPIO.output(green_led,False)
GPIO.output(left_red_led,False)
GPIO.output(right_red_led,False)
# Wait for both buttons to be pressed
while True:
    if (GPIO.input(left_switch)==True and GPIO.input(right_switch)==True):
        break
    time.sleep(.01)
# When we get here both buttons have been pressed
# Turn on the yellow LED
GPIO.output(yellow_led,True)
# Wait five seconds
time.sleep(5)
# Turn on the green LED
GPIO.output(green_led,True)
# Wait for a button to be released
while True:
    if (GPIO.input(left_switch)==False or GPIO.input(right_switch)==False):
        break
    time.sleep(.01)
# When we get here one of the buttons has been released
# Show the winner
if (GPIO.input(left_switch) == False):
    GPIO.output(left_red_led, True)
if (GPIO.input(right_switch) == False):
    GPIO.output(right_red_led, True)
```

This is the complete program, with fancy colouring. If your program doesn't work you might find it useful to compare your program with this version.

## Future Projects

Now that you can read switches and light leds you can do a lot of things. Here are some ideas:

### ***Fixing the Bug***

The game is quite fun, and if people play it correctly it will select the winner each time you play it. However, it is very easy to cheat. If a sneaky player releases the button after the yellow LED has lit but **before** the green LED lights they will win instantly. You can fix this by adding some extra tests so that if anyone has released their button before the green LED is lit the program declares the other player the winner. This will mean adding extra if conditions. Take a look at the ones that decide the winner and see if you can modify those to get the required effect.

### ***Making the game repeat***

At the moment we have to run the program each time we want to play it. It might be nice to have the game play repeatedly. You could add a five second delay and then make the program run again. You can do this by adding another `while True` loop near the top of the program (just before we turn off all the LEDs) and then indenting the rest of the code one level so that all the statements are controlled by the loop.

### ***“Touch the Truck” Game***

There used to be a TV program in Australia called “Touch the Truck”. The idea was that the contestants just had to stay touching a truck, and the last one left touching the truck won it. The game was not very nice and lasted a very long time, with competitors trying desperately to stay awake and keep in the game. You could make a two player “Touch the Truck” game where the first person to release a button loses. You just have to make some simple changes to the loops in this game. You could start by using the `Save As` command to save a new version of the program called perhaps “`touch_the_truck.py`” and then go from there. But don’t play it for too long.

## Hardware and Software Setup

You can do any of the exercises above with a Raspberry Pi using the standard operating system provided by Element 14.

The hardware kits that are used in this exercise, with the breadboard and the cables, can be obtained from SKPang:

<http://www.skpang.co.uk/catalog/starter-kita-for-raspberry-pi-pi-not-include-p-1070.html>

Rob Miles

June 2013

[www.robmiles.com](http://www.robmiles.com)