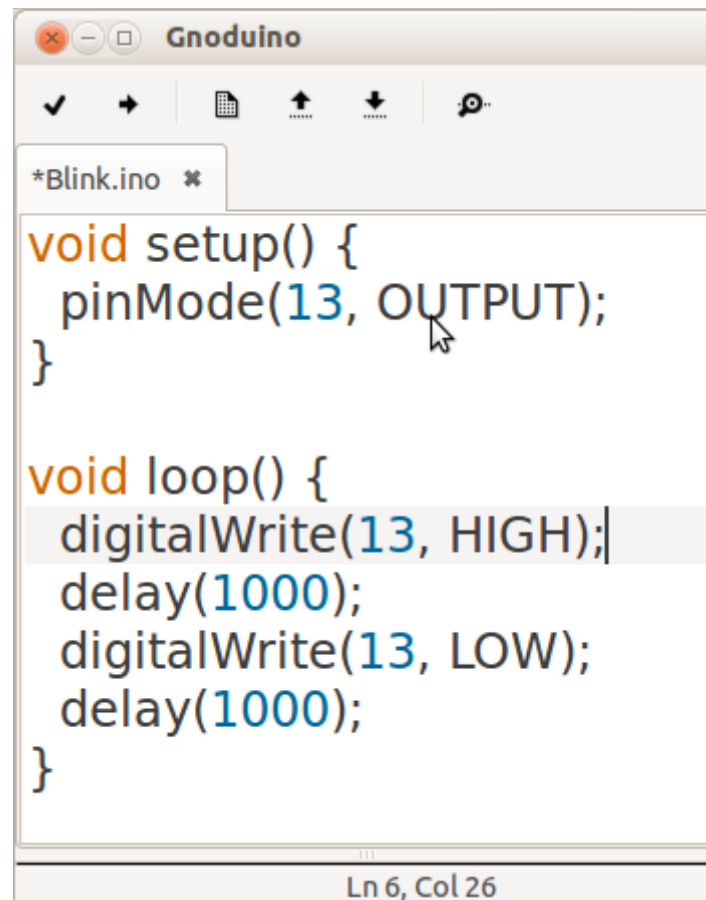


This document summarises some practices pioneered as part of the Shrimping project in Morecambe. To find out more, visit <http://shrimping.it> or Twitter @ShrimpingIt

## Shrimping It : Programming Introduction

To help users in their first encounter with programming, a form of interactive shell programming [ref] is employed, which avoids the need for understanding monolithic code blocks all at once.

To illustrate this, we can look at the classic 'Blink sketch', sample code which is distributed as a simple introductory example for new Arduino users [ref]. It causes the green LED attached to Arduino's pin 13 to toggle between HIGH (light on) and LOW (light off) once a second.

A screenshot of the Gnuino IDE window. The title bar reads "Gnuino". Below the title bar is a toolbar with icons for checkmark, right arrow, file, up arrow, down arrow, and a magnifying glass. A tab labeled "\*Blink.ino\*" is open. The main text area contains the following C++ code:

```
void setup() {  
  pinMode(13, OUTPUT);  
}  
  
void loop() {  
  digitalWrite(13, HIGH);  
  delay(1000);  
  digitalWrite(13, LOW);  
  delay(1000);  
}
```

The code is color-coded: keywords are orange, variables and constants are blue, and strings are grey. A mouse cursor is pointing at the word "OUTPUT" in the first line of the setup function. At the bottom of the window, a status bar shows "Ln 6, Col 26".

To a computer scientist, this code provides an incredibly simple example, and makes perfect sense. By convention, any steps defined in a function called `setup()` are executed only once, whilst the steps in a function called `loop()` are repeated over and over again.

The single step in `setup()` configures pin 13 to be an output. The four steps in `loop()` turn on the LED, wait for a second, turn it off, wait for a second and then repeat in sequence forever.

However, to a novice, a very large number of concepts must be simultaneously understood in order to relate the code to the behaviour in anything more than a superficial way. Although you can read this code somewhat intuitively, to write code for themselves, experimenters need to have total command of every programming detail they use.

Even the first word `void`, and the question ‘why is it there’ can open up a large and complex topic about function definitions, return values and pointers. A shortlist of concepts which are forced upon the reader from just this code listing is roughly as follows; *data types, the void keyword, functional programming with side-effects, function definitions, C statement syntax, code blocks, sequenced arguments.*

Perhaps worse, the only way that a novice programmer can verify their comprehension of any of these aspects is to upload a complete code listing to a #Shrimp, which encapsulates all of the concepts simultaneously, and see if it does what they expect. Frequently this means they are limited to copying whole programs written by others, and learning by modifying small parts. This makes it extremely difficult to create bespoke behaviour for their own project designs, as noone else may have shared code for a sufficiently similar behaviour.

Our teaching approach for novice programmers is very different, and is based on the use of a python library and firmware (pyfirmata and StandardFirmata [ref]) which allows the microcontroller to be remote-controlled from a laptop, one step at a time.

Participants are told we have put software on the #Shrimp which allows us to remote control it. We demonstrate this by issuing single, self-contained commands from the laptop in the python language, using an interactive interpreter known as the python shell [ref].

In this way we interactively introduce and demonstrate the use of individual concepts, in more or less this sequence, values, types, expressions, names, variables, steps, names, variables, functions and loops. After encountering each statement in turn and in isolation, participants are able to confirm their understanding by typing individual steps and observing immediate effects. An example session is illustrated by screenshots and descriptions below.

Of course, it is hard to recreate the interactive presentation here using only screenshots. The reader should note that each line typed is individually explained during the class and when executed they often have immediate and visible consequences.

Each line preceded by `>>>` is a single command, sometimes followed by responses from the computer (lines without the `>>>`).

The sequence is demonstrated at the front of the class using a digital projector, with participants using their own computer, terminal and #Shrimp to demonstrate each of the principles, and experiment with their own variations on the commands we provide.

<pre> &gt;&gt;&gt; 'Shrimp' 'Shrimp' &gt;&gt;&gt; 1 1 &gt;&gt;&gt; 1+1 2 &gt;&gt;&gt; HIGH=1 &gt;&gt;&gt; LOW=0 &gt;&gt;&gt; HIGH 1 &gt;&gt;&gt; </pre>	<pre> &gt;&gt;&gt; from pyfirmata import Arduino &gt;&gt;&gt; shrimp = Arduino('/dev/ttyUSB0') &gt;&gt;&gt; led = shrimp.digital[13] &gt;&gt;&gt; led.write(HIGH) &gt;&gt;&gt; led.write(LOW) &gt;&gt;&gt; from time import sleep &gt;&gt;&gt; sleep(1) &gt;&gt;&gt; sleep(2) &gt;&gt;&gt; sleep(2*2) &gt;&gt;&gt; sleep(0) &gt;&gt;&gt; </pre>	<pre> &gt;&gt;&gt; def flash(): ...     led.write(HIGH) ...     sleep(1) ...     led.write(LOW) ...     sleep(1) ... &gt;&gt;&gt; flash() &gt;&gt;&gt; while True: ...     flash() ... </pre>
<p>Here we introduce values, expressions and types.</p> <p>Initially the computer just parrots the text and number values <code>'Shrimp'</code> and <code>1</code> provided, showing that it recognises them. Then we introduce expressions by showing it can do arithmetic with number values.</p> <p>We show how one or more values can be stored with a human readable name for our convenience. We demonstrate retrieving a named value.</p>	<p>The Arduino IDE tells us our <code>#Shrimp</code> is connected with the name <code>'/dev/ttyUSB0'</code>.</p> <p>In the first line, we load the Arduino remote control functionality from <code>pyfirmata</code>. Connecting with <code>Arduino('/dev/ttyUSB0')</code> we then store the connection for later use with the name <code>shrimp</code>.</p> <p>We get hold of digital pin 13 storing it with the name <code>led</code>. We light the LED on pin 13 with <code>led.write(HIGH)</code>, whilst <code>led.write(LOW)</code> extinguishes it.</p> <p>We load in some time functionality and demonstrate the <code>sleep()</code> command which causes the computer to wait for the specified time in seconds before the cursor reappears. This effect is self-evident during a presentation.</p>	<p>Here a more complex structure is introduced - a function definition - combining the principles already proven by participants themselves.</p> <p>Finally a while loop is introduced, triggering identical behaviour to the earlier Blink sketch.</p> <p>However, unlike the Blink sketch, every constituent part of the program has been introduced and demonstrated individually.</p> <p>As participants ask questions it's often possible to run individual lines of code which correspond to their questions, giving them immediate and concrete answers.</p>

With a final flourish we reveal that all of the individual lines of code we have interactively typed and observed can be put into a file so they can be executed in sequence with a single click - a program, which creates a behaviour that we want.

The listing, `blink.py` looks as follows.

```
from pyfirmata import Arduino
from time import sleep

shrimp=Arduino('/dev/ttyUSB0')
led=shrimp.digital[13]

def flash():
    led.write(1)
    sleep(1)
    led.write(0)
    sleep(1)

while True:
    flash()
```

We can run this stored program by typing

```
python blink.py
```

This is our final step before introducing participants to the full Arduino IDE and the example Blink sketch. Blink.ino is introduced as a way of expressing the same behaviour as Blink.py, but using the language C and the Arduino IDE to put code onto the microcontroller directly. This sequence also allows us to discuss the advantages of having the #Shrimp able to run the behaviour on it's own, without a laptop attached, and the fact that it is a self-contained computer in its own right. We see this as a radically inverted model of disclosure from the standard model adopted as part of Arduino learning.