YTP - Module YTP01

Arduino & Micro-Controllers

Teachers Handbook
LESSON 4
Code – Deeper Dive

Goals & Success Factors

Lesson Goal

- To know your way around the Arduino IDE
    - Toolbars
    - Serial Monitor
    - Compile errors/warnings/messages
- To understand the purpose of functions
    - Notably the Arduino setup() and loop() functions
- To understand basic code constructs
    - Variables
    - Assignments
    - Constants
    - Delays
    - Programming Pins: digitalRead/digitalWrite
    - Comments

Lesson Success Factor(s)

- Understand how to use the Arduino IDE
- Complete basic coding exercises

Runsheet (Guideline)

00:00 Introduction
00:05 Goals & Success Factors
00:10 Recap of last week
00:15 Understanding the Arduino IDE in detail
00:25 Comments, Constants, Variables & delays
00:35 Programming pins (pinMode/read/write)
00:44 Pseudo code examples
01:00 Finish

Extension(s)

Open the Blink sketch
Add code to blink another LED connected to pin 5 once every 2 seconds

Students Pre-work

Watch the following video:
Basic Arduino Coding: bit.ly/ytp01-basic-arduino-coding-1

Teachers Pre-work

Watch the following video:
Arduino Programming Concepts: bit.ly/ytp01-arduino-programming-concepts

The code can be found in the following directory:
https://github.com/nexgencodecamp/ytp/tree/master/modules/arduino/04-coding-deeperdive/handouts

Student Homework

Try and write your own version of the Blink Sketch. In the handouts directory there is a file called homework-challenge.txt to help

Handouts (optional)

Arduino IDE images
Arduino coding tutorial (code)
Homework challenge instructions
Crossword

Please download from:
https://github.com/nexgencodecamp/ytp/tree/master/modules/arduino/04-coding-deeperdive/handouts

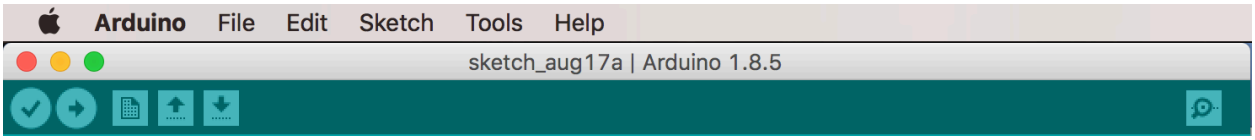
Materials Needed

Arduino IDE

As the intention of this week's lesson is to code, we don't need any special hardware.

Handbook

The Arduino IDE



The Arduino Toolbar (above) is what you will use to typically verify & upload code to your Arduino. The buttons are explained below:

 Verify/Compile code

 Upload code to Arduino

 Save

 Open file

 New file

 Open Serial Monitor

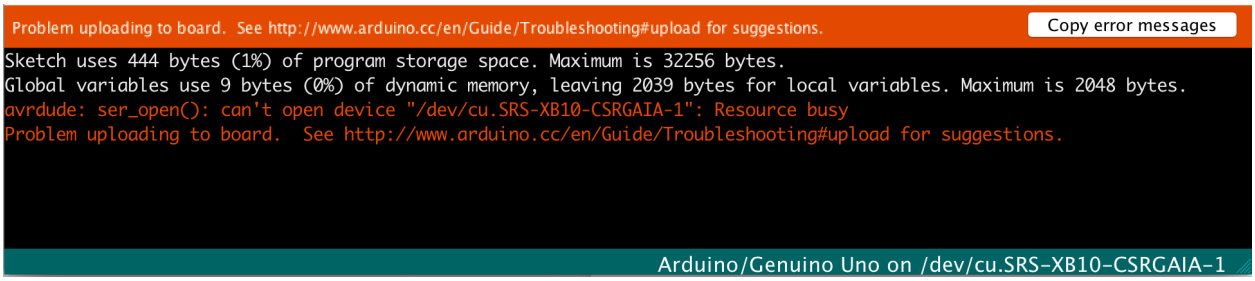
The code window or editor (above) is where you write all your code. At a minimum, every file should contain a setup() function and a loop() function.

Line numbers are shown down the side. They can be turned on and off from the Preferences dialog box. The font size and other editor options can be changed in Preferences.

The name of the file defaults to ‘sketch_’ followed by todays date.

The window at the bottom of the screen is for the display of compiling messages, warning and errors. It is typically blank until you compile or upload some code.

This is what you typically see after compiling a sketch (program). This tells us (amongst other things) that the sketch is 444 bytes in size and lets us know that the maximum size of our code can be 32356 bytes (approximately 32 KiloBytes or 32K)



The above message shows an error explaining that the code cannot be uploaded to the Arduino. It says 'Problem uploading to board'. Sometimes this is easy to track down but not always. It usually comes down to the Arduino not being plugged in or the port not being selected. Sometimes the USB port is 'busy' or unavailable. This often can be cured by unplugging and plugging in the USB cable again.

The Serial Monitor

Sometimes we will want to know what is going on during the running of our program. For this we use the Serial Monitor which is a separate window to which we can write our own messages. For example, let's say we are using an Ultrasonic sensor and we want to know the distance from a solid object whilst we are moving the sensor around. Unless we have a way of outputting the readings to the screen it is almost impossible to know whether the ultrasonic sensor is working properly. This is where the `Serial` object comes in.

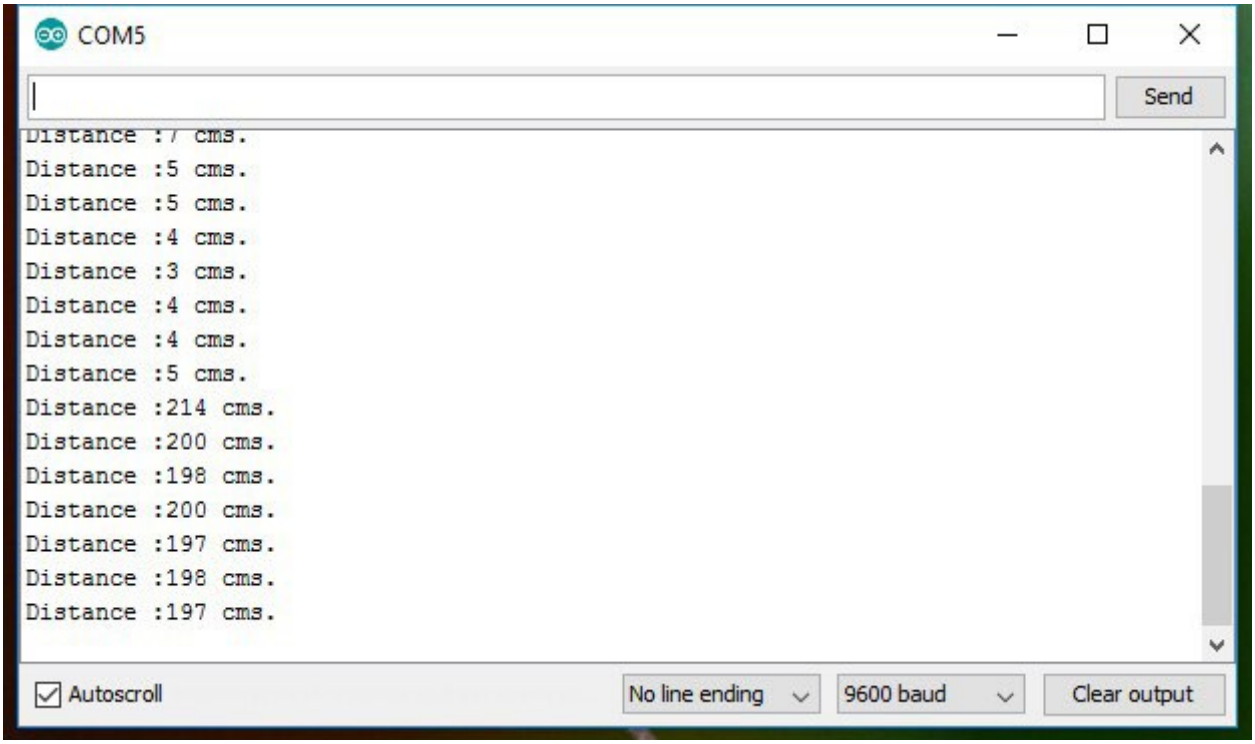
The `Serial` object enables us to write messages to the Serial Monitor. To use it, we first initialise the `Serial` object – typically in the `setup()` function:

```
Serial.begin(9600);
```

Then, in the loop() function we can write messages. Let's say, for instance that we have a variable called distance that every so often we want to check. We could use this code to write to the Serial Monitor:

```
Serial.print("Distance :");
Serial.print(distance);
Serial.print("cms.");
```

If we open the Serial monitor (Tools -> Serial Monitor or clicking the icon), we might see something like this:



What is code?

Code is basically a set of instructions that make up a computer program to accomplish some task. The task we have accomplished so far is to blink a light on and off which conceivably could be part of some useful task or application.

Perhaps a more contrived example could be a program to make toast. If such a thing did exist the instructions or code would go a little like this:

```
1. Find bread
2. If there is no bread, go buy some bread else…
3. Put 2 slices of bread in toaster
4. Set toaster to low/medium/high etc.
5. Turn on toaster
6. If toast is done spread with butter otherwise wait
7. Eat toast
```

These instructions could then be placed into a 'function' – we'll be looking at these below but for now a function is simply an action, some task that contains a set of instructions. The function could look like this:

```
makeToast() {

   1. Find bread
   2. If there is no bread, go buy some bread else…
   3. Put 2 slices of bread in toaster
   4. Set toaster to low/medium/high etc.
   5. Turn on toaster
   6. If toast is done spread with butter otherwise wait
   7. Eat toast

}
```

NB – the first set of brackets () are there to give us a way of passing data into the function. Why would we want to do that? Well imagine that you had to run the above function every time you wanted some toast! You would always end up with 2 slices of buttered toast. Can you see the problem here?

If the makeToast() function is going to be at all useful, we need to be able to be able to make all kinds of toast and any number of slices, toasted to our preference e.g. light/medium/dark. The brackets enable to us pass our preferences 'into' the function so that a better class of function can be realised by taking into account a given set of parameters. The same function but with different parameters might look like this:

```
makeToast(numSlices, strength) {…}
```

or this:

```
makeToast(numSlices, strength, spread) {…}
```

Hopefully, it is beginning to become clearer that functions are very powerful and transform any task into something that can be customised to a particular situation/preference or whatever. If we had stuck with the initial function, things would get very boring. As it is now, we can have practically any kind of toast by passing in either the number of slices and the flavour that we want – much better!

The squiggly brackets are the beginning of the body of the function and inside will lie the instructions. ALL functions must have this analogue (parentheses even if nothing is passed to the function AND squiggly brackets to denote the beginning and the end of the function.

This is how functions are declared/created. If you actually want to *call* a function (and by 'call' we mean actually execute or run the instructions in the function), you just use the name and the parentheses making sure that there are no brackets for the body. Here are a few examples utilising the makeToast function above:

```
// Make the standard 2 slices with butter
makeToast()

// Make 4 slices with butter
makeToast(4)

// Make 2 slices, light with butter
makeToast(4, 'light')

// Make 4 slices, medium with nutella
makeToast(4, 'medium', 'nutella')
```

Hopefully by now we have a reasonable grasp of what we need to do if we want to tackle some task – write a function containing a list of instructions that will try and accomplish the task. We say 'try' because a function will not always do what we hope it will and it may even fail!

Notice that in the code examples above we have some explanation text above each function call prefixed with a double forward slash '//'. This is an example of a comment which we will look at next.

Comments

- Comments are notes either to yourself, your team or any other developer that might read your code over the course of its life.

- Comments should be added to code that may need some explanation

- There are 2 types of comments

    - Single line comments start with a double forward slash '//'. Everything after the forward slash is not code and is therefore bypassed by the compiler when you verify or compile your code.
    - Multiline comments start with a '/*' and end with a '*/'. Everything in between no matter on how many lines is part of the same comment. Multiline comments are useful when you are describing a function or a whole program and therefore require a bit more text than normal.

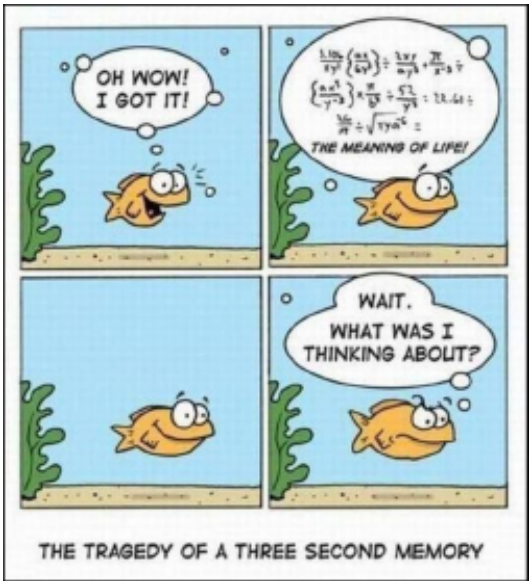
```
// This is a single line comment

/*
      This is a multiline comment. As you can see
      It spans multiple lines.
*/
```

Write some example comments, both single line and multiline of your own in the Arduino editor just to get familiar with the concept.

Variables & Constants

In any software system that models some information, we MUST have the capability of storing data in memory otherwise we would all be like the proverbial goldfish with the mythical 3 second memory.



We store data in variables. These 'variables' can be thought of as buckets that we put data into and retrieve when we need it again. We can also change the data in the bucket (variable) at any time.

An example of a typical variable creation in code for an Arduino program look like the following:

```
int myPin = 2;    // Create an integer called 'myPin' with value = 2
```

Note the comment at the side which explains what the code is doing. We might want to do this because we are going to program pin 2 on the Arduino to control an LED light. Referring to myPin rather than the number 2 gives the program more meaning and is more understandable to someone looking at the code for the first time. After all the number 2 could refer to many things. The value 'myPin' seems like it refers to a pin although granted it could be named even better, say myLEDPin or myLEDOnCheckPin.

A constant is like a variable but once created, it's value can never change. We used a constant in our first Arduino program. It was called LED_BUILTIN – which was its name. It also had a value which happens to be 13. It was probably setup in code like this:

```
#define LED_BUILTIN 13  // Defines a constant called LED_BUILTIN, value = 13
```

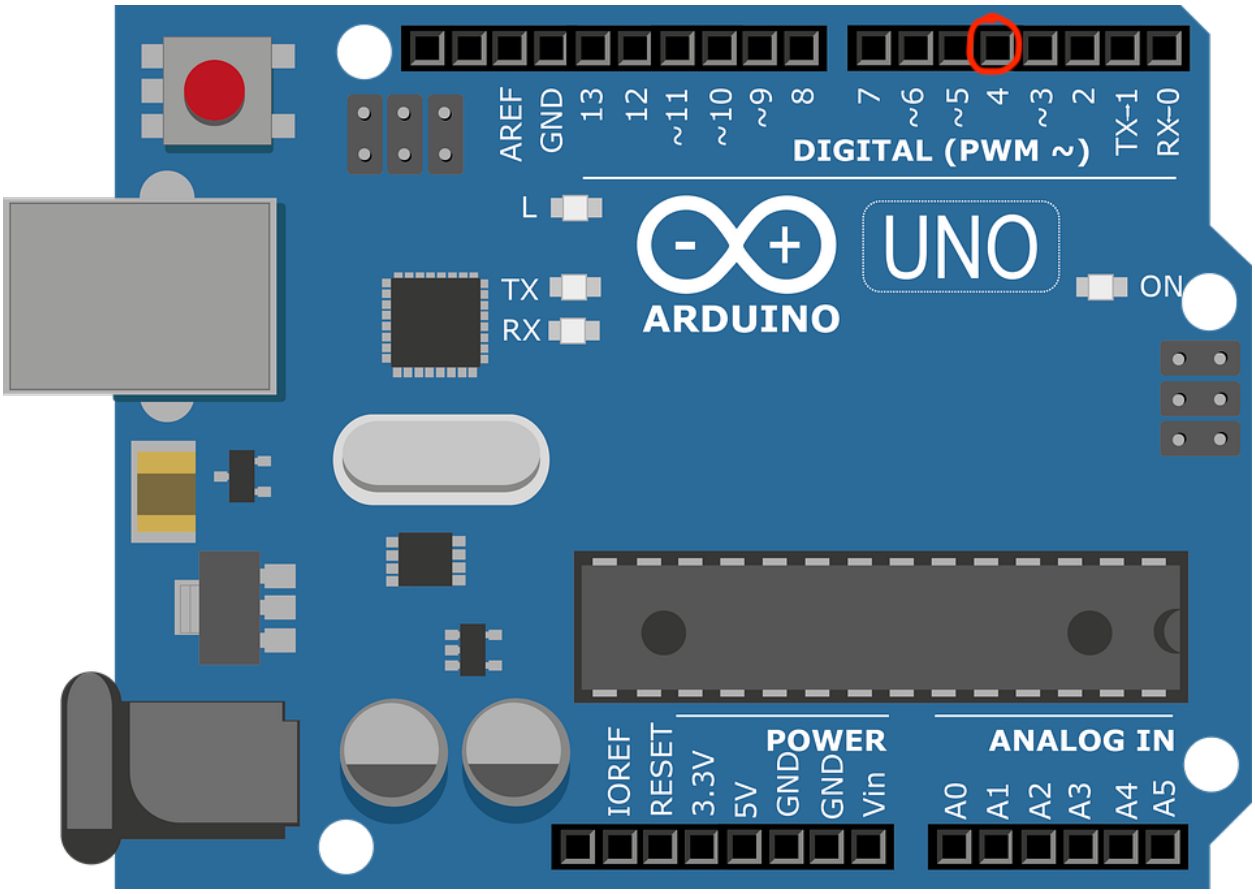
Once again, we have added a comment at the side to explain what is happening in the code.

What if you were asked to do the following. Define a constant called 'MY_LED_PIN' and you knew that it should refer to the actual Arduino digital pin number 7.

How would you define/create this constant using code ?

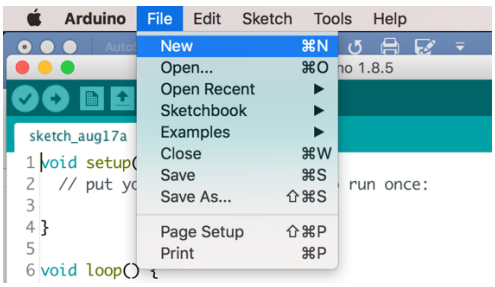
Setting up a PIN in Code

One of the most important things we need to do when using an Arduino for any project is setup one or more pins so that we can program them. Let's say for instance we want to setup and use pin number 4 so that we can connect it to an LED which we will then be able to turn on and off by sending a HIGH or LOW value to it through our code:

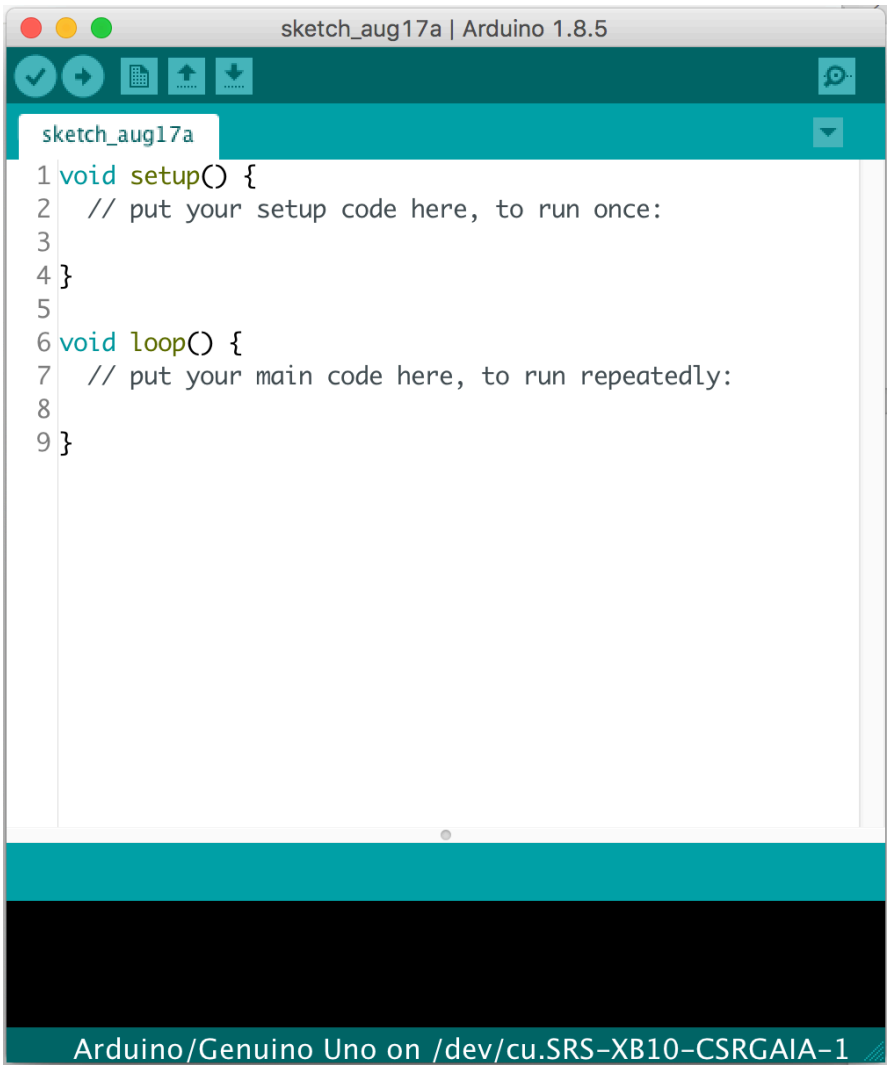


We can see from the Arduino that pin number 4 is a digital pin. If you look at the analogue pins on the other side, you can see that they are numbered A0-A5, thus distinguishing themselves from the digital pins.

Create a new file by starting the Arduino application and then selecting File -> New



You should see the following new window open. If there are any other Arduino windows open, close them.



You *always* start an Arduino program or '**sketch'** as it is called in Arduino-land, with the above 2 functions:

`setup() and loop()`

As you can see, they both have single line comments and nothing much else. The comments are there to instruct you – the programmer – how to proceed. For example: *// put your setup code here, to run once:*

You may recall that functions are actions/verbs/tasks/things that do something/ In the above case, the setup() function configures (or sets up) the environment (pins) and only ever runs once. In fact it runs as soon as the program is run. When it is finished, the loop() function runs not just once but it repeats forever until the Arduino is shut down.
Back to our original problem then. We want to setup pin 4. The first thing we need to decide is:

Is it an INPUT pin or an OUTPUT pin?

Another was of putting it: Do we want to read the pin's value (INPUT) or do we want to set the pin's value (OUTPUT)? Well we have already determined that we are going to use the pin to control an LED i.e. turning the LED on and off. To be able to do that we need to set the pin to be an OUTPUT pin. In doing this, we set the *mode* of the pin and we do it like this:

```
pinMode(4, OUTPUT);
```

The line of code of above literally sets the mode of pin number 4 to output. If we don't initialise a pin in this way, we cannot use it.

Now, how about instead of the number 4, we used an identifier called digPin_4. That would make it a bit clearer to understand what pin 4 is going to be used for. So our final setup would be:

```
pinMode(digPin_4, OUTPUT);
```

Likewise if we wanted to use a pin for reading temperatures, we would set it up as an input pin and so the code would look something this:

```
pinMode(digital_4, INPUT);
```

Functions

We have already talked a lot about functions. They are the guts of any program and really get stuff done. A function is a 'doing' object – it has a single task and hopefully it completes it well every time it is called. We need to understand that coding-wise, you will come across functions in two different contexts:

1. **Context 1 – creating the function OR the function definition**

   The function must be defined or created – I'm sure you will agree. To do this we must use the following syntax. Let's write a function that merely prints out the text 'Hallo'.

```
void sayHello () {
      Serial.print("Hello");
}
```

   The above is the function definition. It establishes the list of instructions that must be carried out if this function is ever called. Nothing will happen until the sayHello() function is actually called. It could stay in the memory of an Arduino forever and never get called, however it still exists.

   Breaking it down…..

   `void` specifies whether the function will return a value to the caller of the function. sayHello() does not return anything but let's say we had a function to add two numbers up together called sum(). It should really return the answer shouldn't it? Thus we would not have the word `void` at the beginning of the function definition, but instead we would put the type of data (in this case a number) that is being returned:

```
int sum (n1, n2) {

      return n1 + n2;

}
```

   Notice that we have a couple of blank lines in the above function. They can be put in purely for aesthetics – i.e. making it easier to read!

   The function name, `sayHello` and `sum` (above) is followed by parentheses – `()`, whose purpose it is to specify whether the function receives any data to help it achieve its task. The `sayHello` function does not receive any data. It doesn't need any to know how to say 'hello'. However the `sum` function receives two pieces of data. If it wasn't given these bits of information, how would it know which numbers to add up ??

   The squiggly brackets mark the beginning and the end of the instructions within the function. They are mandatory and you cannot have one without the other. What falls in

between the squiggly brackets is the guts of the function or the function's code/instructions. If there was nothing in between these two brackets the function would be empty like this:

```
void myEmptyFunction () {
      // Not a lot going on……
}
```

Finally, a function's code is in between the `{` and the `}`. The final squiggly bracket marks the end of the function.

2. **Context 2 – Calling a function**
   Once a function is created, it exists in your program but it will never do what it was made to do until it is 'called'. To call a function you just use the function name followed by parentheses. If the function is expecting some data, the caller may pass the raw data if required.

   For example, our `sayHello` function should be called like this:

   ```
   sayHello()
   ```

   Our sum function on the other hand should be called like this:

   ```
   sum(1, 2)
   ```

   What do you think each function will return ?

Handouts

Handouts are optional but encouraged. They are normally useful for homework but in some cases you may deem them useful for pre-work depending on the upcoming session content. The handouts are shown here but are alternatively available for download at:

https://github.com/nexgencodecamp/ytp/tree/master/modules/arduino/04-coding-deeperdive/handouts

Resources

Basic Arduino Coding
bit.ly/ytp01-basic-arduino-coding-1

Arduino Programming Concepts
bit.ly/ytp01-arduino-programming-concepts