Functions & Abstraction

Programming is problem solving. Typically it involves breaking down a problem into smaller more manageable (and thus easily solvable) pieces. After you go through the process a few times, you'll not only get better and quicker at it, but you'll start to notice a few interesting things about the process.

One is that you tend to be solving the same sort of small problems over and over again. While that's certainly easy, it's boring and not very or efficient. A way to refer to and reuse a previous solution is needed. However, you'll discover that those little problems aren't *exactly* the same, they are similar, but with subtle differences. So building general solutions that are easily adaptable would be preferable.

Another thing you'll discover is that once you've solved a piece and moved on to try to solve the next one, when you need to come back and reuse the first one, you may need to re-learn the details of your original solution in order to use or modify it. Comments and other documentation will help, but if you could just supply a bit of up-front information to your solution to make it changeable, that will save a lot of time.

Fortunately, most programming languages offer tools help with these situations. In our case, the simplest of them is the ***function***. From a syntax standpoint, a function is just a named statement block which can itself be referenced as a statement. And, as it turns out, you've both already used and at least partially created them. All of our programs have used the `setup()` and `loop()` areas, and by adding statements to them, you have been completing the function. All of the various statements to control the GPIO pins, like `pinMode()` and `digitalRead()`, are all actually functions.

Using, known as ***calling***, a function just means referring to it as a command; when you typed `digitalRead(BUTTON)` in your program, you were calling the functioned named digitalRead. The things inside the parenthesis are called ***arguments***, or just ***args*** for short. They are values which the function has access to use during that specific call only. Functions can have 0 args, or many. Just like regular variables, they have to be defined and their type (int, char, float, etc.) defined initially. When there is more than one argument to a function, the order is important. The order used when calling must match the order that is specified in the function definition.

Functions can also ***return*** a value. Hence the usage pattern:

```
x = digitalRead( BUTTON );
```

The digitalRead function takes one argument, which in this case is being sent 'BUTTON', and returns a single value. The *type* of the return value must also be specified in the function definition. Using comments or other documentation to describe what the arguments are for, and what the return value represents is helpful in all but the most simplistic of functions.

Creating your own function is simple. You just need to define the return type, name of the function, and the type and order of any arguments. Then put your statements inside a statement block. Lastly, use the `return` statement to return a value as needed. Okay, that sounds less simple than it is, here's an example:

```
int myMultiply( int x, int y, int z ) {

      int total = x * y * z;
      return total;
}
```

This function, named myMultiply, will return an int value, and it has 3 args, named x, y & z, all also ints. When called it multiplies the passed (x, y & z), and stores the result in another int variable called total. Finally it returns that total variable. So calling it would look like:

```
aValue = myMultiply( 2, 4, 8 );
Serial.println( aValue );
```

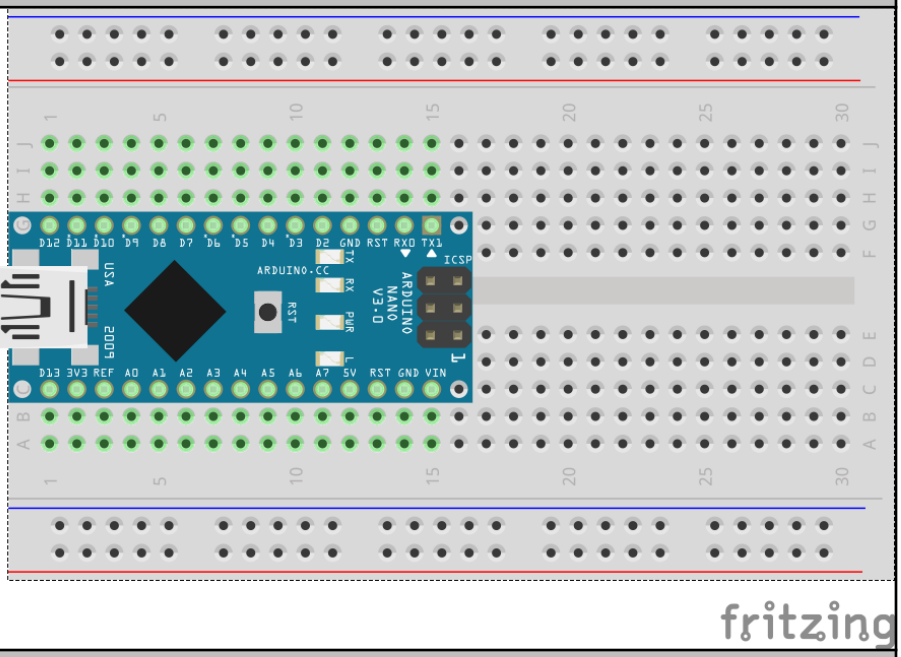
Which outputs the number 64 to the serial console. Note that functions need be created in the global context, that is they cannot be created inside another function like `setup()` or `loop()`.

Programming involves various levels of abstraction. Functions, along with arguments, variables & statements are the basic building blocks that most all programming languages share.

Another more complex logical construct that can greatly enhance the ability to extend and re-use software is the ***Object***. While delving into **OOP (**Object Oriented Programming) is beyond of the scope of this course, just like with functions, you've actually already been using Objects.

The Arduino environment is built on an Object Oriented Language (C++). One of the benefits of the Arduino environment is that the sometimes complex interaction with various external devices can be simplified such that only a few statements are required to setup the device and get information from it. Typically this is done with an Object. For example, the serial link, which we have used a few times now, is accessed via an Object called Serial. We simply write statements which look a lot like function calls, but prefaced with 'Serial.' , the dot being significant, as it tells the compiler to take the following action on the Serial object.We'll explore some object use in the future Experiments.

Experiment: More functionality.

Components	Wiring Diagram
✓Computer ✓Arduino IDE	

Connection Instructions	
Back to just using the Nano and the onboard LED for this one.	
Sketch(es)	funcy.ino

Analysis Questions

Have you been using any functions prior to this section?
Functions return a value, but do other operations?

Programming Tasks

We are still in distress, so we are still trying to blink out SOS, just getting more efficient at it.

Here you are given two functions. One blinks some number of dots, and one some number of dashes.

Call them appropriate inside the loop{} section, along with any other needed statements to get SOS blinking again.

Once that's working, write your own function to reduce the number of statements even further. Make it so that you can call your single function 3 times to blink out SOS.

Hint, your function will need to take two arguments.

Notes

If you need a function which does not return any value at all, define it with the special type of `void`.