A function is a small piece of code that has a name. Writing functions allows us to re-use the same code multiple times across programs. Functions have the the following syntax:

```
function [name of function] {
  # code here
}
```

Pretty simple, right? Let's open up a new file called hello.shso we can write our first simple function.

```
#!/usr/bin/env bash
# File: hello.sh

function hello {
  echo "Hello"
}

hello
hello
hello
```

The entire structure of the function including the function keyword, the name of the function, and the code for the function written inside of the brackets serves as the function definition. The function definition assigns the code within the function to the name of the function (hello in this case). After a function is defined it can be used like any other command. Using our hello command three times should be the equivalent of using echo "Hello" three times. Let's run this script to find out:

```
bash hello.sh

## Hello
## Hello
## Hello
```

It looks like this function works exactly like we expected.

Functions share lots of their behavior with individual bash scripts including how they handle arguments. The usual bash script arguments like $1, $2, and $@ all work within a function, which allows you to specify function arguments. Let's create a slightly modified version of hello.sh which we'll call ntmy.sh:

```
#!/usr/bin/env bash
# File: ntmy.sh

function ntmy {
  echo "Nice to meet you $1"
}
```

In the file above notice that we're not using the ntmy function after we've defined it. That's because we're going to start using the functions that we define as command line programs. So far in this chapter we've been using the syntax of bash [name of script] in order to execute the contents of a script. Now we're going to start using the source command, which allows us to use function definitions in bash scripts as command line commands. Let's use source with this file so that we can then use the ntmy command:

```
source ntmy.sh
ntmy Jeff
ntmy Philip
ntmy Jenny

## Nice to meet you Jeff
## Nice to meet you Philip
## Nice to meet you Jenny
```

And just like that you've created your very own command! Once you close your current shell you'll lose access to the ntmy command, but in the next section we'll discuss how to set up your own commands so that you always have access to them.

Let's write a more complicated function. Imagine that we wanted to add up a sequence of numbers from the command line, but we had no way of knowing how many numbers would be in the sequence. What components would we need to write this function? First we would need a way to capture a list of arguments which can have variable length, second we would need a way to iterate through that list so we could add up each element, and we would need a way to store the cumulative sum of the sequence. These three requirements can be satisfied by using the $@ variable, a FOR loop, and variable where we can store the sum. It's important to break down a larger goal into a series of individual components before writing a program, that way we more easily can identify which features and tools will be required. Let's write this program in a file called addseq.sh.

```
#!/usr/bin/env bash
# File: addseq.sh

function addseq {
  sum=0

  for element in $@
  do
    let sum=sum+$element
  done

  echo $sum
}
```

In the program above we initialize the sum variable to be 0 so that we can add other values in the sequence to sum. We then use a FOR loop to iterate through every element of $@, which is an array of all the arguments we provide to addseq. Finally we echo the value of sum. Let's source

this program and test it out:

```
source addseq.sh
addseq 12 90 3
addseq 0 1 1 2 3 5 8 13
addseq
addseq 4 6 6 6 4

## 105
## 33
## 0
## 26

```

By breaking down a large problem we were able to write a nice little function!

Mark as completed