Review: Functions

This is a review of what we covered in this tutorial on functions.

We often want to be able to re-execute blocks of code when we are writing programs, without having to re-write the block of code entirely. We need a way of grouping code together and giving it a name, so that we can call it by that name later, and that's what we call a ***function***.

To create a function, we must first declare it and give it a name, the same way we'd create any variable, and then we follow it by a function definition:

```
var sayHello = function() {
};
```

We could put any code inside that function - one statement, multiple statements - depends on what we want to do. In this function, we could just output a message at a random location:

```
var sayHello = function() {
   text("Hallllllllo!", random(200), random(200));
};
```

Now, if all we do is declare the function, nothing will happen. In order for the program to execute the code that's inside the function, we actually have to "call" the function, by writing its name followed by empty parentheses:

```
sayHello();
```

And then we could call it whenever we wanted, as many times as we wanted!

```
sayHello();
sayHello();
sayHello();
```

We often want to be able to customize functions, to tell the program "well, do all of this code, but change a few things about how you do it." That way we have code that is both reusable *and* flexible, the best of both worlds. We can achieve that by specifying "arguments" for a function, using those arguments to change how the function works, and passing them in when we call the function.

For example, what if we wanted to be able to say exactly where we want the message displayed, just like we can say exactly where we want to draw rect()s and ellipse()s? We could imagine calling it like so, to put the message at two precise coordinates:

```
sayHello(50, 100);
sayHello(150, 200);
```

To make that work, we need to change our `sayHello` function definition so it knows that it will receive 2 arguments, and then uses them inside:

```
var sayHello = function(xPos, yPos) {
   text("Hallllllllo!", xPos, yPos);
};
```

The arguments that get passed in basically become like variables inside your function definition, and the names depend on what you call them in the parentheses. We could just as easily rename them to something shorter:

```
var sayHello = function(x, y) {
```

```
   text("Hallllllllo!", x, y);
};
```

Our functions can accept any number of arguments - zero, one, two, or more. We could have also decided that we wanted to change our function to accept a name to say hello to:

```
var sayHello = function(name) {
   text("Hallllllllo, " + name, random(200), random(200));
};
```

And then we would have called it like so:

```
sayHello("Winston");
sayHello("Pamela");
```

We could combine those ideas, and have it accept three arguments, for the name and position:

```
var sayHello = function(name, x, y) {
   text("Hallllllllo " + name, x, y);
};
```

And then call it like so:

```
sayHello("Winston", 10, 100);
```

It really depends on what you want your functions to do, and how much you want to customize what they can do. You can always start off with no arguments, and then add more as you realize you need them.

Now, you've actually been calling functions this whole time- that's how you've been making drawing and animations - like with `rect`, `ellipse`, `triangle`, etc. All of those functions are ones that come from the ProcessingJS library, and we load them into every program that you make here, so that you can always use them. We've defined the functions for you, because we thought they'd be useful, and now it's up to you to decide what custom functions you want to use in your own programs. For example, we provide the `ellipse` function, but we don't provide a `cat` function - if your program involves a lot of different cats in different locations, maybe you should create your own cat function!

There's another powerful thing we can do with functions - we can use them to take in some values, compute them, and return a new value. Think about all the things you can do with a calculator - add values, subtract, calculate square root, multiply, etc. All of those would be done with functions that took in the input and output the result. The functions would take in the input as arguments and output the result using a ***return statement***. Here's a function that adds two numbers and returns the result:

```
var addNumbers = function(num1, num2) {
  var result = num1 + num2;
  return result;
};

var sum = addNumbers(5, 2);
text(sum, 200, 200); // Displays "7"
```

The ***return statement*** does two things: it gives a value back to whoever called it (which is why we could store it in the `sum` variable), and it immediately exits the function. That means it'd be silly if we had something like this, because that last line would never get executed:

```
var addNumbers = function(num1, num2) {
  var result = num1 + num2;
  return result;
```

```
  result = result * 2; // silly!
};
```

Functions with return values are quite useful for manipulating data in programs, and they can be combined together in expressions, too:

```
var biggerSum = addNumbers(2, 5) + addNumbers(3, 2);
```

You can even call functions inside function calls, although that can get hard to read at times:

```
var hugeSum = addNumbers(addNumbers(5, 2), addNumbers(3, 7));
```

Now that you know how to create functions that wrap around blocks of code, we have to bring up an important concept: ***local variables*** versus ***global variables***.

When we declare a new variable *inside* a function, we say that it is local to that function. That's because only that function can see that variable - the rest of the program outside of it cannot. Once we're outside that function, it's like it no longer exists. In the following function, `localResult` is a local variable:

```
var addNumbers = function(num1, num2) {
  var localResult = num1 + num2;
  println("The local result is: " + localResult);
  return localResult;
};
addNumbers(5, 7);
println(localResult); // oh noes!
```

When we run that code, we'll get an error on the final line: "localResult is not defined." The variable is only defined inside the function, because that's where we declared it with the `var localResult =` line, and is not defined outside of it.

When we declare a variable outside our functions, we say that it is a global variable. That's because all functions can now access it and do whatever they want with it.

```
var globalResult;

var addNumbers = function(num1, num2) {
  globalResult = num1 + num2;
  println("The global result is: " + globalResult);
};
addNumbers(5, 7);
println(globalResult);
```

When we run the above code, we will not get an error, because we declared `globalResult` outside of the function, so we can access it wherever we want.

Every programming language is different, but for JavaScript, it's important to know that variables have "function scope" - a function can see the local variables that were declared inside of it and the global variables that were declared outside of it, but it cannot see the local variables inside other functions.