SuperCollider Tutorial

Chapter 2

By Celeste Hutchins

2005

www.celesteh.com

Functions

Nicole feels like she's got the hang of SuperCollider and she heard Bush say that the economy is picking up, so she's dropped out of grad school to work for the new SuperCollider start-up company SuperSounds.com. On her first day, her boss tells her to write a function that prints "hello world" four times. "No problem," she thinks and goes to look at her class notes. **Functions** are code blocks encased by curly brackets, { } and hello world is easy enough. So she writes:

```
(
	{

		var greet;
		greet = "hello world";

		greet.postln;
		greet.postln;
		greet.postln;
		greet.postln;
	}
)
```

Then she thinks, "I should have asked for a signing bonus." She tries running it, by double clicking to the right of the top parenthesis and hitting enter. In the Untitled output window it says, `"a Function"`

What's the problem? She declared a variable called `greet`. `greet` gets `"hello world"`. Then she sends a `postln` message to `greet` four times. Every line ends with a semicolon . . .

Then she realizes that she defined a function, but never told the interpreter to run it. The interpreter saw a code block surrounded by curly brackets and thought "a function!" Then it thought, "What do you want me to do with this? Nothing? Ok, I'll throw it away." So Nicole modifies her code:

```
(
	var func;

	func = {

		var greet;
		greet = "hello world";

		greet.postln;
		greet.postln;
		greet.postln;
		greet.postln;
	};

	func.value;
)
```

And then it works great. `value` is a message you can send to functions. It means, "Run yourself."

But then Nicole gets a message from her boss saying, "Sometimes, we need to print out hello world five times and once in a while, three times, and rarely, it needs to print out infinite times." Nicole considers writing a few different versions of the function and calling them func4, func3, etc, but then remembers about **arguments** to functions.

```
(
	var func;
	func = { arg repeats;

		var greet;
		greet = "hello world";

		repeats.do ({
			greet.postln;
		});
	};

	func.value(4);
)
```

When she writes her function, she **declares** to the interpreter that the function takes one argument. An argument is a special type of variable that gets set when the function is called. When she calls the function with `func.value(4);`, she's assigning a value to the argument `repeats`.

Then, inside the function, she's written, `repeats.do`. What is 'do'? It's a message. It takes a function as an argument. 'do' is a message you can send to **integers** that runs the function passed as an argument the number of times as the integer that it was called on. In this example, repeats is 4, so it runs four times.

What is an integer? An integer is a whole number. -2, -1, 0, 1, 2, 3, 4, etc. There is a special integer in SuperCollider called inf. It means infinity. If we try calling our above function, with

```
func.value(inf);
```

hello world will print out forever, or until we stop the program by hitting apple-. .

Then Nicole's boss sends another email saying that marketing has some changes. Every line needs to start out with the line number, starting with zero. So she makes another change:

```
(
     var func;
     func = { arg repeats;

          var greet;
          greet = "hello world";

          repeats.do ({ arg index;
               index.post;
               " ".post;
               greet.postln;
          });
     };

     func.value(4);
)
```

The function called by do takes an argument. And that argument is the number of times the loop has been run, starting with 0. So her output from this program is:

```
0 hello world
1 hello world
2 hello world
3 hello world
```

`post` just means print without a new line at the end.

Almost every time, she runs this function, the argument is going to be 4. So she can declare a default argument for the function.

```
(
	var func;
	func = { arg repeats = 4;

		var greet;
		greet = "hello world";

		repeats.do ({ arg index;
			index.post;
			" ".post;
			greet.postln;
		});
	};

	func.value;
)
```

When she calls `func.value`, if there's no argument, then the interpreter will assign 4 to repeats by default. If she calls it with, `func.value(6);` then repeats gets 6 instead. What if she accidentally passes in something besides an Integer into her function? That depends. If the object she passes in also understands a do message, then it will use that instead, although the result may differ. Otherwise, she will get an error.

What if the function took a lot of arguments with default values?

```
(
	var func;
	func = { arg foo = 0, bar = 0, baz = 1, repeats = 4;

		var greet;
		greet = "hello world";

		repeats.do ({ arg index;
			index.post;
			" ".post;
			greet.postln;
		});
	};

	func.value;
)
```

If she wants to pass in arguments, she does it in the order they're declared in.

```
func.value(0 /* foo */, 0 /* bar */, 1 /* baz */, 3 /* repeats */);
```

However, if we're happy with all the default values, there's a way to tell the function just to assign to a particular variable, out of order:

```
func.value(repeats: 3);
```

You can also just pass in the first N arguments and leave the remainder to the default values:

```
func.value(3, 1);
```

And you can combine approaches:

```
func.value(2, repeats: 6);
```

Those slash-stars up there are **comments**. The interpreter ignores everything between a forward slash star and a backslash star. They can span multiple lines. You can also create single line comment by using forward slash forward slash:

```
// This is a comment
```

Some times it's useful to **comment out** a line of code while **debugging**. So you can skip a particular line in order to figure out where your error is.

The philosophical point of a function is to **return** a value. Doing things within functions, like printing are technically called **side effects**. What the function returns is the value of its last line. Let's change that function so it returns the number of times that it printed.

```
(
	var func;
	func = { arg repeats = 4;

		var greet;
		greet = "hello world";

		repeats.do ({ arg index;
			index.post;
			" ".post;
			greet.postln;
		});
```

```
        repeats;
    };

    func.value;
)
```

Now, if we create a new variable called `times`, we can assign the output of the function to it.

```
(
    var func, times;
    func = { arg repeats = 4;

        var greet;
        greet = "hello world";

        repeats.do ({ arg index;
            index.post;
            " ".post;
            greet.postln;
        });
        repeats;
    };

    times = func.value;
    times.postln;
)
```

Prints out hello world with the line number like before, and then at the bottom, prints out a 4. Or we could change those last two lines from

```
    times = func.value;
```

```
	times.postln;
```

to

```
func.value.postln;
```

and the interpreter will read that statement left to right, first finding the value of the function and then sending that value a postln message.

Ok, what happens if we take the above function and write some code below it that looks like this:

```
(
	var func, times;
	func = { arg repeats = 4;

		var greet;
		greet = "hello world";

		repeats.do ({ arg index;
			index.post;
			" ".post;
			greet.postln;
		});
		repeats;
	};
	greet.postln;
)
```

We get errors.

- ERROR: Variable 'greet' not defined.
  in file 'selected text'
  line 14 char 6 :
    greet•.postln;

This is because of something called **scope**. Variables only exist in the code block in which they are declared. Code blocks are zero or more lines of code surrounded by parenthesis, or by curly braces. This means that variables and arguments declared inside a function only exist inside that function. `index` does not exist outside of its function, which is the function passed as an argument to `repeats.do`. `greet` does not exist outside of it's function. None of these variables exist outside of the parenthesis.

Variables in outer blocks are accessible within inner blocks. For example,

```
(
	var func, times;
	times = 0;
	func = { arg repeats = 4;

		var greet;
		greet = "hello world";
		times.postln;

		repeats.do ({ arg index;
			index.post;
			" ".post;
			greet.postln;
		});
		repeats;
	};
)
```

Is fine because `times` exists in the outer most block. In the same way, we can use greet inside our repeats.do function.

There are, however, a few variables that can be used everywhere. The interpreter gives us 26 **global variables**. Their scope is all of SuperCollider. They are single letter variable names, a, b, c, d, e, f, etc. You don't need to declare them and they keep their value until you change it again, even in different code blocks. This is why the variable 's' refers to the Server. You can change that, but you might not want to.

If you have any questions about functions, you can look at the functions help file for more information. The most useful message you can send to a function is value, but there are a few others, which you can read about. You may not understand everything you see in the helpfiles, but it's good to keep reading them.

Numbers and Math

We just learned about Integers, which, remember are whole numbers, like -1, 0, 1, 2. And we learned about the do message.

What are some things you might want to do to a number? Add, subtract, multiply, divide, modulus.

Remember algebra where a*b + c*d = (a * b) + (c * d). And remember how that one cheap calculator just did what you punched in, in the order you punched it in, without respecting order of operations? SuperCollider is like that cheap calculator. Mathematical expressions are evaluated left to right. Addition and subtraction have the same **precedence** as multiplication and division. That

means that if you want to do things in some order, you've got to use parenthesis.

Parentheses are evaluated innermost to outermost.

(a + (b * (c + d)))
(2 + (4 * (3 + 2))) = (2 + (4 *(5))) = (2 + (20)) = 22;

One math operation that you might not have seen before is **modulus**. It means "remainder" and it's represented by a '%'.

10 % 3 = 1
26 % 9 = 8

Ok, so the output of a modulus between two integers is an integer, by definition. And if you add or subtract two integers, you get an integer. As you do if you multiply two integers. But what is the result of division?

3 / 2 = 1.5

1.5 is not an integer. It's a type of number called a Floating point, or in SuperCollider, a **Float**. A float is a fraction. 1.1, 2.5, -0.2, 5.0 are all Floats. They also can add subtract, etc.

The number story so far:

- Whole numbers are Integers
- Real numbers are Floats
- Indicate order of operations with parenthesis
- Math operations are evaluated innermost to outermost and left to right
- Modulus (%) means remainder

You can do more with numbers than simple math and do loops. There are many interesting and useful messages one can pass to Integer. It has a help file worth reading. So type in the word Integer, starting with a capital-I and press apple-shift-? to look at the help file. The top of that help file says:

superclass: SimpleNumber

superclass is a vocabulary word. It refers to a concept called **inheritance**. What this superclass designation means is that Integer **is a** SimpleNumber. When you define classes (recall that a call is the definition for an object), you can define **subclasses** of any class. A subclass is a special type of the original class. It **inherits** all the properties of its **superclass**. So the subclass is the child and the superclass is the parent. We'll come back to this. But what it means for us is that Integer **is a** SimpleNumber. Which means it understands all the messages that you can pass to a SimpleNumber. So to find out about what these inherited messages are, we should highlight "SimpleNumber" and hit apple-shift-? Float and Integer both inherit from SimpleNumber, so take a look at that help file. Looking at help files and trying stuff out will get you learning the language faster than anything else. They might not make a lot of sense right now, but if you keep looking, you'll get context and be able to figure them out in the future.

What about order of operations at other times? If we have `func.value(3+4)`, it evaluates `3+ 4` to `7` before calling the function. We can put anything we want inside those parentheses and it will evaluate what's inside until it gets to the outermost parenthesis and then it will call the function.

We have one more example that hopefully ties all of this together. Remember our SynthDef from Chapter 1?

```
(

	var syn, sound;

	syn = SynthDef.new("example2", {arg freq = 440, amp = 0.2;
		Out.ar(0, SinOsc.ar(freq, mul: amp));
	});

	syn.load(s);
)
```

We've added a field called amp, which indicates the **amplitude** or volume at which the sound should play.

Now, lets' write some code to play N overtones of 100Hz. We need to make sure that our overall amplitude doesn't exceed 1, or else we'll get peaking. So, we will divide 1 by the number of overtones into 1 to get the amp value to send to each synth.

```
(
	var func;

	func = { arg repeats = 4;
		repeats.do({ arg index;

			Synth.new("example2", [\freq, (index + 1) * 100,
					\amp, 1 / repeats]);
		});
	};
	func.value;
)
```

Let's hear what happens when we play this. All the sounds play at once. This is because the loop goes through as fast as it can. Next time, we'll learn how to pause a loop and write a SynthDef that stops playing by itself.

Problems

1. Translate the following algebraic expressions into proper supercollider syntax, using parenthesis where needed. You answers should be easily understandable, even to people who don't know SC's order of operations.
   1.1. 3 + 5 * 4
   1.2. 3 * 2 + 1
   1.3. 3 * (4 + 2)
   1.4. 2 * 3 + 4 * 5 + 7 / 2 + 1

2. Get yourself familiar with modulus by working out these problems by hand:
   2.1. 5 % 4
   2.2. 163 % 9
   2.3. 20 % 5
   2.4. 17 % 6,
   2.5. 23 % 2
   2.6. 3 % 5

3. Write a function to print out the first *n* multiples of 10, starting from 0. Pass the number of multiples into the function as an argument. Set the default number to 7.

4. `rand` is a message that you can send to numbers. `y.rand` returns a

number between 0 and y. Humans are generally able to hear frequencies between 20Hz and 20000Hz (the freq argument is Hz). Write a function to play *n* random pitches and the first *m* overtones of those pitches. Make sure that the random pitches are all in the audible range. Set the amplitude of the overtones to one half of the amplitude of the fundamentals. Pass the number of random pitches and the number of overtones into the function as arguments. Set the default number of random pitches to 2 and the default number of overtones to 3. Make certain that your total amplitude will not peak.

4.1. Can you figure out a way to make sure that all your pitches, including overtones, are in the audible range?