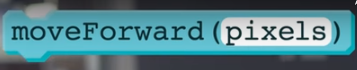
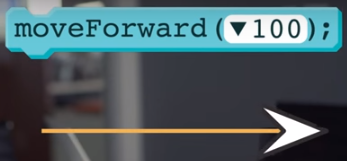
Name \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Date \_\_\_\_\_\_\_\_\_\_\_\_ Period \_\_\_\_\_\_\_\_\_

**Unit 3 Lesson 8 Functions with Parameters**

You’ve already seen that some functions are defined with parameters to make them more \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ useful.  For example: the original version of moveForward() moved the turtle exactly 25 pixels. This didn’t give you a lot of options for how to design your drawings. A much more useful version of moveForward is one that accepts a parameter that allows you to specify exactly how far forward the turtle should move.

Up to this point the functions *we* have written ourselves don’t have parameters, but the ability to write your own functions with parameters is both useful and powerful.  So let’s see how to do it.

Here’s an example: look at this function that draws a square with sides 100 pixels long.

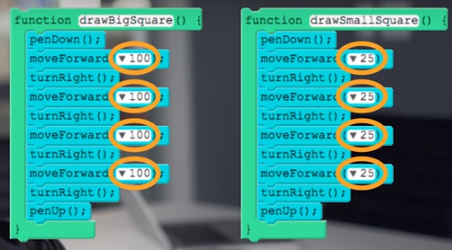
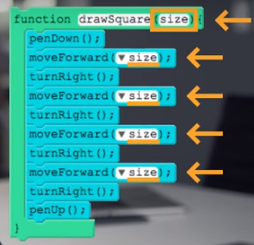
If we want to make a function to draw a square with  sides 50 pixels long, we’d have to make a new function that’s almost identical to our first one. The only difference is how far the turtle is moving forward each time.

And if we wanted even more options for the size of square we draw, every time we will need to declare a new function. Yeah that is not very good. But luckily, there’s better way. And there is….

The solution to this problem is the ability to *create* a function that accepts a “parameter”. A \_\_\_\_\_\_\_\_\_\_\_\_\_\_ is a named value that is provided as input to a function.

We saw that for our versions of drawSquare a lot of the code was repeated - the turtle was executing the same basic behavior, just moving by different amounts.

It turns out that we can express this generalized \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ in our code as well. Here’s a new version of the drawSquare. Here we’ve made the size of the square a parameter of the function.

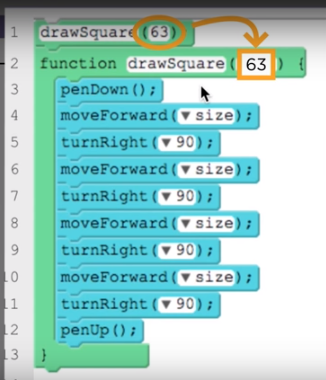
 = 

Notice that the function heading \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ that it needs an \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.  We’ve given a name to that input, and that name is *\_\_\_\_\_\_\_\_\_\_\_\_\_.* And everywhere in the code we want to use that \_\_\_\_\_\_\_\_\_\_, we just enter the word “size” instead of a \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.

Writing a function with a parameter is very similar to writing a function without one.  Just drag out the function block that shows a parameter and drop it in the workspace.  We add the name of our parameter inside the \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ in the function call, like this.

Just as giving a meaningful and descriptive name to your function is important, so is deciding what to \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ a parameter - the name of the parameter should give some insight into what the \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ will be used for.  In this case, we’ve chosen the name “size” to indicate that you can set the size of the square when calling the function. As a programmer, what you call your parameter is your decision to make.

We can write the body of the function like this…The parameter - size - acts as a \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ for a value.  Wherever we want to use that value, we just refer to it as ‘size’ instead, of a number.

Let’s look at what happens when the function is called.  Now when a user calls drawSquare they call it with a value like this:  drawSquare(63).

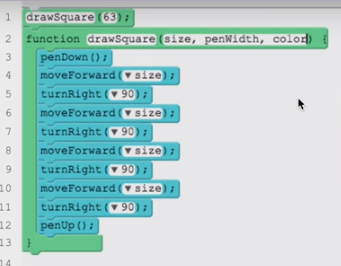
We say that the value - 63 - “gets \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_” to the parameter of the function.  The parameter acts like a \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ with a name on it. Whatever value is passed to the function is stored in that container.

Inside the body of the function you will want to use the value \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ in the parameter.

In order to do so you refer to it by the \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ you gave your parameter. As the program is running whenever the word size is encountered it is \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ by the value stored in the parameter.

Adding a parameter to our drawSquare allows us to \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ its functionality. In other words, our function is useful in an entire \_\_\_\_\_\_\_\_\_\_\_\_\_\_ of situations, as opposed to just being useful in \_\_\_\_\_\_\_\_\_ particular instance. The result is a function that gives you much more \_\_\_\_\_\_\_\_\_\_\_\_\_\_ and control, without the need to create duplicated code.

Once you have added a single parameter to a function you might realize that there are multiple aspects of its functionality you would like to generalize.

For our square example you might want to change the width or color of the line. To do this you can simply add more parameters and separate them by \_\_\_\_\_\_\_\_\_\_\_\_\_\_. You can then use any of these parameter names in the body of your function. When calling your function you’ll need to include multiple \_\_\_\_\_\_\_\_\_\_\_, one for each parameter, and each one separated by commas.

Parameters are another example of how \_\_\_\_\_\_\_\_\_\_\_\_ can be used to solve problems. When confronted with a group of related problems we first work to identify the \_\_\_\_\_\_\_\_\_ across all of them. In doing so you might realize that using parameters would let you define a *single* function with general behavior that can solve any instance of the problem with one piece of code. This a powerful skill that \_\_\_\_\_\_\_\_\_\_\_\_\_ the need for creating duplicated code, and leads to programs that are easier to read, write, and maintain.