

3.10: Functions

Now that we are comfortable creating some code, let us take a look at how to do more with what we have. In any language we use, there are concepts that are not always translatable from one to the next, if the concept is even present in both. In spoken languages, words for a particular concept also may have no direct translation to another language. For example, the German word *Kummerspeck* is a single word to convey the concept of excess weight gained by emotion-related overeating. Not only does the English language lack a word defining this concept, its closest literal translation in English would be “grief bacon.” Similarly, in Wagiman (Australia) there is an infinitive murr-ma, which means “to walk along in the water searching for something with your feet,”¹ which is both much more specific and requires a significantly longer English sentence to convey.

The development of words that specify such lengthy or concise ideas arise out of the popularity of, or need to convey that concept often in a society. It is far easier to say “I am going to the gym to get rid of this Kummerspeck” than “I am going to the gym to get rid of the weight I gained from overeating due to my emotions,” which allows for the concept to be used more often.

In programming languages, we can find a parallel to this when functions or abilities in one language are not available in another, or require additional code to achieve. For example, in PHP you can retrieve items available to the page using the GET method by referencing the built-in `$_GET` array:

```
1. <?php echo $_GET['variable name']; ?>
```

Attempting to implement the same functionality in JavaScript requires much more effort, even with the assistance of its built-in `location.search` function:

```
1. var $_GET = {},
2. variablesList = location.search.substr(1).split(/&/);
3. for (var x=0; x<variablesList.length; ++i) {
4.   var tmp = variablesList[x].split(/=/);
5.   if (tmp[0] != "") {
6.     $_GET[decodeURIComponent(tmp[0])] = decodeURIComponent(tmp.slice(1).join("").replace("+", " "));}}
```

The programming equivalent of creating a word to describe a concept is the ability to create our own algorithms and functions, to describe new actions. In the example above, we could take our JavaScript code and wrap it in a function definition, and add a return call to send the results back to where you called the function, like this:

```
1. function findValues(){
2.   var $_GET = {},
3.   variablesList = location.search.substr(1).split(/&/);
4.   for (var x=0; x<variablesList.length; ++i) {
5.     var tmp = variablesList[x].split(/=/);
6.     if (tmp[0] != "") {
7.       $_GET[decodeURIComponent(tmp[0])] = decodeURIComponent(tmp.slice(1).join("").replace("+", " "));
8.     }
9.   }
10.  return $_GET;
11. }
```

Now, anytime you wanted to find your values using JavaScript, you could include this function and simply type:

```
1. var $_GET = findValues(); document.write($_GET['variable name'];
```

Creating a function also allows us to reference it in more than one place, without having to retype or copy and paste those lines into every place we want to use them. This also means that debugging only requires fixing the original block of code in your function, as it is the only place the line exists, since each function call is a reference to this single definition.

Creating a function in PHP is much like the example we looked at for JavaScript above. To create a function, we use the word `function` (denoting that we are creating one, not calling one to use it) followed by the name we want to give it and any variables we would like it to use in parenthesis. A set of braces are used to identify the code that belongs to the definition. An empty function called `Add` that does not accept any variables would look like this:

```
1. function Add(){ }
```

Note that a terminating semi-colon is not needed after the closing brace. The brace tells PHP that the statement is complete, just as it does when using a logic or control statement. To pass variables into our Add function, the function will need to know what to expect to receive. We do this by adding parameters (names we give to the variables passed to the function) to the definition. Here we will add two parameters:

```
1. function Add($num1, $num2){ }
```

Now, we will tell the function to add these numbers and give us the result:

```
1. $var1 = 4; $var2= 5;
2. function Add($num1, $num2){
3. $temp = $num1 + $num2;
4. return $temp;
5. }
6. $value = Add($var1, $var2);
```

When we use the function and pass it actual values (in this example, \$var1 and \$var2) those variables are called arguments; they contain the data we actually want to use. Another example of function output you may see are ones that send output right to the screen, like this:

```
1. $var1=4; $var2=5;
2. function Add($num1, $num2){
3. $temp = $num1 + $num2;
4. print $temp;
5. }
6. Add($var1, $var2);
```

They might also output more than we expect, not just the result of the equation:

```
1. function Add($num1, $num2){
2. $temp = $num1 + $num2;
3. print "$num1 + $num2 = $temp";
4. $oddEven = $temp % 2;
5. if ($oddEven == 0){ print "<br/>$temp is even"; }
6. else{ print "<br/>$temp is odd"; }
7. }
8. Add(7,9);
```

While all of these example are effective, the second two examples actually limit our ability to use them again, namely by performing multiple actions, and by mixing action and output. Since the power of a function lies largely in our ability to reuse it, these are attributes we will want to eliminate. To make functions as useful as possible, they should do one thing, and do it well.

In this example the intent of our function is to add numbers. Our expectation is that we will provide it numbers, and it will provide the sum. Any other actions or steps needed to solve a larger problem should reside elsewhere in your code, or in another function. By doing this, we know that when we use add() all it will do is give us a number. If the function shows our output instead of returning it, we would not be able to use it if we did not want it to show on the screen. By returning it and storing it, we can choose where and when it is displayed, or use it somewhere else.

To simplify our function and follow these practices, let us refine it to the following:

```
1. function Add($num1, $num2){ return $num1 + $num2; }
```

Now we can call the function and store it to a variable:

```
1. $sum = Add(3,5);
```

Or we would chose to display it on the screen:

```
1. echo Add(3,5);
```

Useful Feature

Wait! Where did the \$temp variable go?! By skipping the use of declaring \$temp to hold our sum, we have eliminated one reserved memory space while the function is running. In terms of the size of this function, the number of variables we needed, and the time for it to execute, we would never know the difference. However, in much larger programs, especially when memory is limited, these steps can improve performance.

Let us take a look at how we can put all of this together:

```
1. <?php
2. function Add($num1, $num2){
3. return $num1 + $num2;
4. }
5. echo "Welcome to the number adding page! <br/>";
6. echo "The sum of 3 and 5 is " . Add(3,5);echo "<br/>The sum of 9 and 12 is " . Add(9, 12);
7. ?>
```

Seeing as the ability to add numbers is already built into PHP, let us look at a more advanced example. We will design a function that tells us how much to add to a bill for a tip. To do this, we will need to create a function that takes in the total of our bill, calculates the tip amount, and tells us how much the tip would be.

Remember, a function should do one thing only, so before we worry about final totals or any other issues, let us at least get that far:

```
1. function tip($billTotal){
2. $tip = $billTotal * .15;
3. return $tip;
4. }
```

In the above example, we assumed a 15% tip. What if that is not the standard where you are? Or maybe the service was better or worse than you expected. This is an additional use case we need to consider. To do this, we will allow the user to tell us the percentage they want, and assume that if they do not tell us, it is still 15%:

Additional notes

Do not forget your SCOPE! Functions cannot see anything outside of their braces that are not given to them when they are called. Once you leave your function, any local variables created in it that were not returned are gone!

```
1. function tip($billTotal, $percent=.15){
2. $tip = $billTotal * $percent;
3. return $tip;
4. }
```

Setting \$percent equal to a value in the function definition tells the function that if a second value is not passed, assume it to be .15. This allows that variable to be optional, so we can call this function as tip(45.99,.20) or just tip(45.99). We are still only doing one thing, but now we have some flexibility. What about other use cases, like splitting the bill? While we could tell the function how many people to divide the total by, that would violate our “one thing” rule. Instead, we can divide the total outside of the function, and give tip() the result to figure out how much each person’s contribution to the tip would be.

If you have been testing out our function as we have progressed, you have probably had some tip values that resulted in half pennies and even smaller fractions of currency to fulfill. This is because our function does not know we are dealing with money—it is just doing the math. Since we only need to calculate to the second decimal point for this type of problem, we can round out our answer to match. PHP already provides us with a rounding function called round, so we will use that to refine our response:

```
1. function tip($billTotal, $percent=.15){
2. $tip = $billTotal * $percent;
3. $roundedTip = round($tip, 2);
4. return $roundedTip;
5. }
```

Lastly we will combine our statements, as we did before, to eliminate extra variables and shorten our code:

```
1. function tip($billTotal, $percent=.15){ return round(($billTotal * $percent),2) }
```

Now we have a concise function that can perform a single task and perform it well. We can refer to it anywhere in our page as often as we need to without having to copy and paste the code. In fact, this is the perfect time to introduce a set of functions that allow us to import other files into our page.

Additional notes

Order helps! If all of your optional variables are at the end (right hand side) of the definition, you will not have to pass empty quotes as place holders if you only want to send the required variables.

The functions `include()`, `require()`, and `require_once()` all allow us to pass a file location on our server that we want to use as part of our code. By importing these outside files, we can create libraries of functions or class files and insert them into any page we want to use them on. Each of these functions runs the same way, by passing a file location. If the contents of the file are not critical to your page, you may want to just use `include`, as it will not generate an error if the file is missing. Errors will be thrown by using `require()` or `require_once()`. The former will always insert the contents of the file, while the latter will only load the file if its contents are not already available. Using `required_once()` will save us from redefinition errors caused by redefining functions or classes we already have. If the contents of the file `tip.php` was our function, we could reference it in any page like this:

```
1. <?php
2. require_once("tip.php");
3. echo tip(49.99);
4. ?>
```

Learn more

Keywords, search terms: Functions, function scope

Tizag's function review: <http://www.tizag.com/phpT/phpfunctions.php>

Helper Functions : <http://net.tutsplus.com/tutorials/php/increase-productivity-by-creating-php-helper-functions/>

3.10: Functions is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

- **3.10: Functions** by [Michael Mendez](#) has no license indicated.