Chapter 5. Functions

When you're writing computer programs, laziness is a virtue. Reusing code you've already written makes it easier to do as little work as possible. Functions are the key to code reuse. A *function* is a named set of statements that you can execute just by invoking the function name instead of retyping the statements. This saves time and prevents errors. Plus, functions make it easier to use code that other people have written (as you've discovered by using the built-in functions written by the authors of the PHP interpreter).

The basics of defining your own functions and using them are laid out in Section 5.1. When you call a function, you can hand it some values with which to operate. For example, if you write a function to check whether a user is allowed to access the current web page, you would need to provide the username and the current web page name to the function. These values are called *arguments*. Section 5.2 explains how to write functions that accept arguments and how to use the arguments from inside the function.

Some functions are one-way streets. You may pass them arguments, but you don't get anything back. A `print_header( )` function that prints the top of an HTML page may take an argument containing the page title, but it doesn't give you any information after it executes. It just displays output. Most functions move information in two directions. The access control function mentioned above is an example of this. The function gives you back a value: `true` (access granted) or `false` (access denied). This value is called the *return value*. You can use the return value of a function like any other value or variable. Return values are discussed in Section 5.3.

The statements inside a function can use variables just like statements outside a function. However, the variables inside a function and outside a function live in two separate worlds. The PHP interpreter treats a variable called `$name` inside a function and a variable called `$name` outside a function as two unrelated variables. Section 5.4 explains the rules about which variables are usable in which parts of your programs. It's important to understand these rules — get them wrong and your code relies on uninitialized or incorrect variables. That's a bug that is hard to track down.

5.1 Declaring and Calling Functions

To create a new function, use the `function` keyword, followed by the function name and then, inside curly braces, the function body. Example 5-1 declares a new function called `page_header( )`.[1]

[1] Strictly speaking, the parentheses aren't part of the function name, but it's good practice to include them when referring to functions. Doing so helps you to distinguish functions from variables and other language constructs.

Example 5-1. Declaring a function

```
function page_header( ) {
    print '<html><head><title>Welcome to my site</title></head>';
    print '<body bgcolor="#ffffff">';
}
```

Function names follow the same rules as variable names: they must begin with a letter or an underscore, and the rest of the characters in the name can be letters, numbers, or underscores. The PHP interpreter doesn't prevent you from having a variable and a function with the same name, but you should avoid it if you can. Many things with similar names makes for programs that are hard to understand.

The `page_header( )` function defined in Example 5-1 can be called just like a built-in function. Example 5-2 uses `page_header( )` to print a complete page.

Example 5-2. Calling a function

```
page_header( );
print "Welcome, $user";
print "</body></html>";
```

Functions can be defined before or after they are called. The PHP interpreter reads the entire program file and takes care of all the function definitions before it runs any of the commands in the file. The `page_header( )` and `page_footer( )` functions in Example 5-3 both execute successfully, even though `page_header( )` is defined before it is called and `page_footer( )` is defined after it is called.

Example 5-3. Defining functions before or after calling them

```
function page_header( ) {
    print '<html><head><title>Welcome to my site</title></head>';
    print '<body bgcolor="#ffffff">';
}

page_header( );
print "Welcome, $user";
page_footer( );

function page_footer( ) {
    print '<hr>Thanks for visiting.';
    print '</body></html>';
}
```

.2 Passing Arguments to Functions

While some functions (such as `page_header( )` in the previous section) always do the same thing, other functions operate on input that can change. The input values supplied to a function are called *arguments*. Arguments add to the power of functions because they make functions more flexible. You can modify `page_header( )` to take an argument that holds the page color. The modified function declaration is shown in Example 5-4.

Example 5-4. Declaring a function with an argument

```
function page_header2($color) {
    print '<html><head><title>Welcome to my site</title></head>';
    print '<body bgcolor="#' . $color . '">';
}
```

In the function declaration, you add `$color` between the parentheses after the function name. This lets the code inside the function use a variable called `$color`, which holds the value passed to the function when it is called. For example, you can call the function like this:

```
page_header2('cc00cc');
```

This sets `$color` to `cc00cc` inside `page_header2( )`, so it prints:

```
<html><head><title>Welcome to my site</title></head><body bgcolor="#cc00cc">
```

When you define a function that takes an argument as in Example 5-4, you must pass an argument to the function when you call it. If you call the function without a value for the argument, the PHP interpreter complains with a warning. For example, if you call `page_header2( )` like this:

```
page_header2( );
```

The interpreter prints a message that looks like this:

```
PHP Warning:  Missing argument 1 for page_header2( )
```

To avoid this warning, define a function to take an optional argument by specifying a default in the function declaration. If a value is supplied when the function is called, then the function uses the supplied value. If a value is not supplied when the function is called, then the function uses the default value. To specify a default value, put it after the argument name. Example 5-5 sets the default value for `$color` to `cc3399`.

Example 5-5. Specifying a default value

```
function page_header3($color = 'cc3399') {
    print '<html><head><title>Welcome to my site</title></head>';
    print '<body bgcolor="#' . $color . '">';
}
```

Calling `page_header3('336699')` produces the same results as calling `page_header2('336699')`. When the body of each function executes, `$color` has the value `336699`, which is the color printed out for the `bgcolor` attribute of the `<body>` tag. But while `page_header2( )` without an argument produces a warning, `page_header3( )` without an argument runs just fine, with `$color` set to `cc3399`.

Default values for arguments must be literals, such as `12`, `cc3399`, or `Shredded Swiss Chard`. They can't be variables. The following is not OK:

```
$my_color = '#000000';

// This is incorrect: the default value can't be a variable.
function page_header_bad($color = $my_color) {
    print '<html><head><title>Welcome to my site</title></head>';
    print '<body bgcolor="#' . $color . '">';
}
```

To define a function that accepts multiple arguments, separate each argument with a comma in the function declaration. In Example 5-6, `page_header4( )` takes two arguments: `$color` and `$title`.

Example 5-6. Defining a two-argument function

```
function page_header4($color, $title) {
    print '<html><head><title>Welcome to ' . $title . '</title></head>';
    print '<body bgcolor="#' . $color . '">';
}
```

To pass a function multiple arguments when you call it, separate the argument values by commas in the function call. Example 5-7 calls `page_header4( )` with values for `$color` and `$title`.

Example 5-7. Calling a two-argument function

```
page_header4('66cc66','my homepage');
```

Example 5-7 prints:

```
<html><head><title>Welcome to my homepage</title></head><body bgcolor="#66cc66">
```

In Example 5-6, both arguments are mandatory. You can use the same syntax in functions that take multiple arguments to denote default argument values as you do in functions that take one argument. However, all of the optional arguments must come after any mandatory arguments. Example 5-8 shows the correct ways to define a three-argument function that has one, two, or three optional arguments.

Example 5-8. Multiple optional arguments

```
// One optional argument: it must be last
function page_header5($color, $title, $header = 'Welcome') {
    print '<html><head><title>Welcome to ' . $title . '</title></head>';
    print '<body bgcolor="#' . $color . '">';
    print "<h1>$header</h1>";
}
// Acceptable ways to call this function:
page_header5('66cc99','my wonderful page'); // uses default $header
page_header5('66cc99','my wonderful page','This page is great!'); // no defaults

// Two optional arguments: must be last two arguments
function page_header6($color, $title = 'the page', $header = 'Welcome') {
    print '<html><head><title>Welcome to ' . $title . '</title></head>';
    print '<body bgcolor="#' . $color . '">';
    print "<h1>$header</h1>";
}
// Acceptable ways to call this function:
page_header6('66cc99'); // uses default $title and $header
page_header6('66cc99','my wonderful page'); // uses default $header
page_header6('66cc99','my wonderful page','This page is great!'); // no defaults


// All optional arguments
function page_header6($color = '336699', $title = 'the page', $header = 'Welcome') {
    print '<html><head><title>Welcome to ' . $title . '</title></head>';
    print '<body bgcolor="#' . $color . '">';
    print "<h1>$header</h1>";
```

```
}
// Acceptable ways to call this function:
page_header7( ); // uses all defaults
page_header7('66cc99'); // uses default $title and $header
page_header7('66cc99','my wonderful page'); // uses default $header
page_header7('66cc99','my wonderful page','This page is great!'); // no defaults
```

All of the optional arguments must be at the end of the argument list to avoid ambiguity. If `page_header7( )` could be defined with a mandatory first argument of `$color`, an optional second argument of `$title`, and a mandatory third argument of `$header`, then what would `page_header7('33cc66','Good Morning')` mean? The '`Good Morning`' argument could be a value for either `$title` or `$header`. Putting all optional arguments after any mandatory arguments avoids this confusion.

Any changes you make to a variable passed as an argument to a function don't affect the variable outside the function. In Example 5-9, the value of `$counter` outside the function doesn't change.

Example 5-9. Changing argument values

```
function countdown($top) {
    while ($top > 0) {
        print "$top..";
        $top--;
    }
    print "boom!\n";
}

$counter = 5;
countdown($counter);
print "Now, counter is $counter";
```

Example 5-9 prints:

```
5..4..3..2..1..boom!
Now, counter is 5
```

Passing `$counter` as the argument to `countdown( )` tells the PHP interpreter to copy the value of `$counter` into `$top` at the start of the function, because `$top` is the name of the argument. Whatever happens to `$top` inside the function doesn't affect `$counter`. Once the value of `$counter` is copied into `$top`, `$counter` is out of the picture for the duration of the function.

Modifying arguments doesn't affect variables outside the function even if the argument has the same name as a variable outside the function. If `countdown( )` in Example 5-9 is changed so that its argument is called `$counter` instead of `$top`, the value of `$counter` outside the function doesn't change. The argument and the variable outside the function just happen to have the same name. They remain completely unconnected.

5.3 Returning Values from Functions

The header-printing function you've seen already in this chapter takes action by displaying some output. In addition to an action such as printing data or saving information into a database, functions can also compute a value, called the *return value*, that can be used later in a program. To capture the return value of a function, assign the function call to a variable. Example 5-10 stores the return value of the built-in function `number_format( )` in the variable `$number_to_display`.

Example 5-10. Capturing a return value

```
$number_to_display = number_format(285266237);
print "The population of the US is about: $number_to_display";
```

Just like Example 1-6, Example 5-10 prints:

```
The population of the US is about: 285,266,237
```

Assigning the return value of a function to a variable is just like assigning a string or number to a variable. The statement `$number = 57` means "store `57` in the variable `$number`." The statement `$number_to_display = number_format(285266237)` means "call the `number_format( )` function with the argument `285266237` and store the return value in `$number_to_display`." Once the return value of a function has been put into a variable, you can use that variable and the value it contains just like any other variable in your program.

To return values from functions you write, use the `return` keyword with a value to return. When a function is executing, as soon as it encounters the `return` keyword, it stops running and returns the associated value. Example 5-11 defines a function that returns the total amount of a restaurant check after adding tax and tip.

Example 5-11. Returning a value from a function

```
function restaurant_check($meal, $tax, $tip) {
    $tax_amount = $meal * ($tax / 100);
    $tip_amount = $meal * ($tip / 100);
    $total_amount = $meal + $tax_amount + $tip_amount;

    return $total_amount;
}
```

The value that `restaurant_check( )` returns can be used like any other value in a program. Example 5-12 uses the return value in an `if( )` statement.

Example 5-12. Using a return value in an if( ) statement

```
// Find the total cost of a $15.22 meal with 8.25% tax and a 15% tip
$total = restaurant_check(15.22, 8.25, 15);

print 'I only have $20 in cash, so...';
if ($total > 20) {
    print "I must pay with my credit card.";
} else {
```

```
    print "I can pay with cash.";
}
```

A particular `return` statement can only return one value. You can't return multiple values with something like `return 15, 23`. If you want to return more than one value from a function, you can put the different values into one array and then return the array.

Example 5-13 shows a modified version of `restaurant_check( )` that returns a two-element array containing the total amount before the tip is added and after it is added.

Example 5-13. Returning an array from a function

```
function restaurant_check2($meal, $tax, $tip) {
    $tax_amount  = $meal * ($tax / 100);
    $tip_amount  = $meal * ($tip / 100);
    $total_notip = $meal + $tax_amount;
    $total_tip   = $meal + $tax_amount + $tip_amount;

    return array($total_notip, $total_tip);
}
```

Example 5-14 uses the array returned by `restaurant_check2( )`.

Example 5-14. Using an array returned from a function

```
$totals = restaurant_check2(15.22, 8.25, 15);

if ($totals[0] < 20) {
    print 'The total without tip is less than $20.';
}
if ($totals[1] < 20) {
    print 'The total with tip is less than $20.';
}
```

Although you can only return a single value with a `return` statement, you can have more than one `return` statement inside a function. The first `return` statement reached by the program flow inside the function causes the function to stop running and return a value. This isn't necessarily the `return` statement closest to the beginning of the function. Example 5-15 moves the cash-or-credit-card logic from Example 5-12 into a new function that determines the appropriate payment method.

Example 5-15. Multiple return statements in a function

```
function payment_method($cash_on_hand, $amount) {
    if ($amount > $cash_on_hand) {
        return 'credit card';
    } else {
        return 'cash';
    }
}
```

Example 5-16 uses `payment_method( )` by passing it the result from `restaurant_check( )`.

Example 5-16. Passing a return value to another function

```
$total = restaurant_check(15.22, 8.25, 15);
$method = payment_method(20, $total);
print 'I will pay with ' . $method;
```

Example 5-16 prints the following:

```
I will pay with cash.
```

This is because the amount `restaurant_check( )` returns is less than 20. This is passed to `payment_method( )` in the `$total` argument. The first comparison in `payment_method( )`, between `$amount` and `$cash_on_hand`, is `false`, so the code in the `else` block inside `payment_method( )` executes. This causes the function to return the string `cash`.

The rules about truth values discussed in Chapter 3 apply to the return values of functions just like other values. You can take advantage of this to use functions inside `if( )` statements and other control flow constructs. Example 5-17 decides what to do by calling the `restaurant_check( )` function from inside an `if( )` statement's test expression.

Example 5-17. Using return values with if( )

```
if (restaurant_check(15.22, 8.25, 15) < 20) {
    print 'Less than $20, I can pay cash.';
} else {
    print 'Too expensive, I need my credit card.';
}
```

To evaluate the test expression in Example 5-17, the PHP interpreter first calls the `restaurant_check( )` function. The return value of the function is then compared with `20`, just as it would be if it were a variable or a literal value. If `restaurant_check( )` returns a number less than 20, which it does in this case, then the first `print` statement is executed. Otherwise, the second `print` statement runs.

A test expression can also consist of just a function call with no comparison or other operator. In such a test expression, the return value of the function is converted to `true` or `false` according to the rules outlined in Section 3.1. If the return value is `true`, then the test expression is `true`. If the return value is `false`, so is the test expression. A function can explicitly return `true` or `false` to make it more obvious that it should be used in a test expression. The `can_pay_cash( )` function in Example 5-18 does this as it determines whether we can pay cash for a meal.

Example 5-18. Functions that return true or false

```
function can_pay_cash($cash_on_hand, $amount) {
    if ($amount > $cash_on_hand) {
        return false;
    } else {
        return true;
    }
}
```

```
$total = restaurant_check(15.22,8.25,15);
if (can_pay_cash(20, $total)) {
    print "I can pay in cash.";
} else {
    print "Time for the credit card.";
}
```

In Example 5-18, the `can_pay_cash( )` function compares its two arguments. If `$amount` is bigger, then the function returns `true`. Otherwise, it returns `false`. The `if( )` statement outside the function single-mindedly pursues its mission as an `if( )` statement — finding the truth value of its test expression. Since this test expression is a function call, it calls `can_pay_cash( )` with the two arguments: `20` and `$total`. The return value of the function is the truth value of the test expression and controls which message is printed.

Just like you can put a variable in a test expression, you can put a function's return value in a test expression. In any situation where you call a function that returns a value, you can think of the code that calls the function, such as `restaurant_check(15.22,8.25,15)`, as being replaced by the return value of the function as the program runs.

One frequent shortcut is to use a function call with the assignment operator in a test expression and to rely on the fact that the result of the assignment is the value being assigned. This lets you call a function, save its return value, and check whether the return value is `true` all in one step. Example 5-19 demonstrates how to do this.

Example 5-19. Assignment and function call inside a test expression

```
function complete_bill($meal, $tax, $tip, $cash_on_hand) {
    $tax_amount = $meal * ($tax / 100);
    $tip_amount = $meal * ($tip / 100);
    $total_amount = $meal + $tax_amount + $tip_amount;
    if ($total_amount > $cash_on_hand) {
        // The bill is more than we have
        return false;
    } else {
        // We can pay this amount
        return $total_amount;
    }
}

if ($total = complete_bill(15.22, 8.25, 15, 20)) {
    print "I'm happy to pay $total.";
} else {
    print "I don't have enough money. Shall I wash some dishes?";
}
```

In Example 5-19, the `complete_bill( )` function returns `false` if the calculated bill, including tax and tip, is more than `$cash_on_hand`. If the bill is less than or equal to `$cash_on_hand`, then the amount of the bill is returned. When the `if( )` statement outside the function evaluates its test expression, the following things happen:

1. `complete_bill( )` is called with arguments `15.22`, `8.25`, `15`, and `20`.
2. The return value of `complete_bill( )` is assigned to `$total`.

3. The result of the assignment (which, remember, is the same as the value being assigned) is converted to either `true` or `false` and used as the end result of the test expression.

5.4 Understanding Variable Scope

As you saw in Example 5-9, changes inside a function to variables that hold arguments don't affect those variables outside of the function. This is because activity inside a function happens in a different *scope*. Variables defined outside of a function are called *global variables*. They exist in one scope. Variables defined inside of a function are called *local variables*. Each function has its own scope.

Imagine each function is one branch office of a big company, and the code outside of any function is the company headquarters. At the Philadelphia branch office, co-workers refer to each other by their first names: "Alice did great work on this report," or "Bob never puts the right amount of sugar in my coffee." These statements talk about the folks in Philadelphia (local variables of one function), and say nothing about an Alice or a Bob who works at another branch office (local variables of another function) or at company headquarters (global variables).

Local and global variables work similarly. A variable called `$dinner` inside a function, whether or not it's an argument to that function, is completely disconnected from a variable called `$dinner` outside of the function and from a variable called `$dinner` inside another function. Example 5-20 illustrates the unconnectedness of variables in different scopes.

Example 5-20. Variable scope

```
$dinner = 'Curry Cuttlefish';

function vegetarian_dinner( ) {
    print "Dinner is $dinner, or ";
    $dinner = 'Sauteed Pea Shoots';
    print $dinner;
    print "\n";
}

function kosher_dinner( ) {
    print "Dinner is $dinner, or ";
    $dinner = 'Kung Pao Chicken';
    print $dinner;
    print "\n";
}

print "Vegetarian ";
vegetarian_dinner( );
print "Kosher ";
kosher_dinner( );
print "Regular dinner is $dinner";
```

Example 5-20 prints:

```
Vegetarian Dinner is , or Sauteed Pea Shoots
Kosher Dinner is , or Kung Pao Chicken
Regular dinner is Curry Cuttlefish
```

In both functions, before `$dinner` is set to a value inside the function, it has no value. The global variable `$dinner` has no effect inside the function. Once `$dinner` is set inside a function, though, it doesn't affect the global `$dinner` set outside any function or the `$dinner` variable in another function. Inside each function, `$dinner` refers to the local version of `$dinner` and is completely separate from a variable that happens to have the same name in another function.

Like all analogies, though, the analogy between variable scope and corporate organization is not perfect. In a company, you can easily refer to employees at other locations; the folks in Philadelphia can talk about "Alice at headquarters" or "Bob in Atlanta," and the overlords at headquarters can decide the futures of "Alice in Philadelphia" or "Bob in Charleston." With variables, however, you can access global variables from inside a function, but you can't access the local variables of a function from outside that function. This is equivalent to folks at a branch office being able to talk about people at headquarters but not anyone at the other branch offices, and to folks at headquarters not being able to talk about anyone at any branch office.

There are two ways to access a global variable from inside a function. The most straightforward is to look for them in a special array called `$GLOBALS`. Each global variable is accessible as an element in that array. Example 5-21 demonstrates how to use the `$GLOBALS` array.

Example 5-21. The $GLOBALS array

```
$dinner = 'Curry Cuttlefish';

function macrobiotic_dinner( ) {
    $dinner = "Some Vegetables";
    print "Dinner is $dinner";
    // Succumb to the delights of the ocean
    print " but I'd rather have ";
    print $GLOBALS['dinner'];
    print "\n";
}

macrobiotic_dinner( );
print "Regular dinner is: $dinner";
```

Example 5-21 prints:

```
Dinner is Some Vegetables but I'd rather have Curry Cuttlefish
Regular dinner is: Curry Cuttlefish
```

Example 5-21 accesses the global `$dinner` from inside the function as `$GLOBALS['dinner']`. The `$GLOBALS` array can also modify global variables. Example 5-22 shows how to do that.

Example 5-22. Modifying a variable with $GLOBALS

```
$dinner = 'Curry Cuttlefish';

function hungry_dinner( ) {
    $GLOBALS['dinner'] .= ' and Deep-Fried Taro';
}

print "Regular dinner is $dinner";
```

```
print "\n";
hungry_dinner( );
print "Hungry dinner is $dinner";
```

Example 5-22 prints:

```
Regular dinner is Curry Cuttlefish
Hungry dinner is Curry Cuttlefish and Deep-Fried Taro
```

Inside the `hungry_dinner( )` function, `$GLOBALS['dinner']` can be modified just like any other variable, and the modifications change the global variable `$dinner`. In this case, `$GLOBALS['dinner']` has a string appended to it using the concatenation operator from Example 2-19.

The second way to access a global variable inside a function is to use the `global` keyword. This tells the PHP interpreter that further use of the named variable inside a function should refer to the global variable with the given name, not a local variable. This is called "bringing a variable into local scope." Example 5-23 shows the `global` keyword at work.

Example 5-23. The global keyword

```
$dinner = 'Curry Cuttlefish';

function vegetarian_dinner( ) {
    global $dinner;
    print "Dinner was $dinner, but now it's ";
    $dinner = 'Sauteed Pea Shoots';
    print $dinner;
    print "\n";
}

print "Regular Dinner is $dinner.\n";
vegetarian_dinner( );
print "Regular dinner is $dinner";
```

Example 5-23 prints:

```
Regular Dinner is Curry Cuttlefish.
Dinner was Curry Cuttlefish, but now it's Sauteed Pea Shoots
Regular dinner is Sauteed Pea Shoots
```

The first `print` statement displays the unmodified value of the global variable `$dinner`. The `global $dinner` line in `vegetarian_dinner( )` means that any use of `$dinner` inside the function refers to the global `$dinner`, not a local variable with the same name. So, the first `print` statement in the function prints the already-set global value, and the assignment on the next line changes the global value. Since the global value is changed inside the function, the last `print` statement outside the function prints the changed value as well.

The `global` keyword can be used with multiple variable names at once. Just separate each variable name with a comma. For example:

```
global $dinner, $lunch, $breakfast;
```

Generally, I recommend that you use the `$GLOBALS` array to access global variables inside functions instead of the `global` keyword. Using `$GLOBALS` provides a reminder on every variable access that you're dealing with a global variable. Unless you're writing a very short function, it's easy to forget that you're dealing with a global variable with `global` and become confused as to why your code is misbehaving. Relying on the `$GLOBALS` array requires a tiny bit of extra typing, but it does wonders for your code's intelligibility.

You may have noticed something strange about the examples that use the `$GLOBALS` array. These examples use `$GLOBALS` inside a function, but don't bring `$GLOBALS` into local scope with the `global` keyword. The `$GLOBALS` array, whether used inside or outside a function, is always in scope. This is because `$GLOBALS` is a special kind of pre-defined variable, called an *auto-global* . Auto-globals are variables that can be used anywhere in your PHP programs without anything required to bring them into scope. They're like a well-known employee that everyone, at headquarters or a branch office, refers to by his first name.

The auto-globals are always arrays that are automatically populated with data. They contain things such as submitted form data, cookie values, and session information. Chapter 6 and Chapter 8 each describe specific auto-global variables that are useful in different contexts.

5.5 Chapter Summary

Chapter 5 covers:

- Defining your own functions and calling them in your programs.
- Defining a function with mandatory arguments.
- Defining a function with optional arguments.
- Returning a value from a function.
- Understanding variable scope.
- Using global variables inside a function.

5.6 Exercises

1. Write a function to print out an HTML `<img>` tag. The function should accept a mandatory argument of the image URL and optional arguments for `alt` text, height, and width.
2. Modify the function in the previous exercise so that the filename only is passed to the function in the URL argument. Inside the function, prepend a global variable to the filename to make the full URL. For example, if you pass *photo.png* to the function, and the global variable contains `/images/`, then the `src` attribute of the printed `<img>` tag would be `/images/photo.png`. A function like this is an easy way to keep your image tags correct, even if the images move to a new path or a new server. Just change the global variable — for example, from `/images/` to `http://images.example.com/`.

3. What does the following code print out?

```
  $cash_on_hand = 31;
  $meal = 25;
  $tax = 10;
  $tip = 10;
  while(($cost = restaurant_check($meal,$tax,$tip)) < $cash_on_hand) {
      $tip++;
      print "I can afford a tip of $tip% ($cost)\n";
  }
  function restaurant_check($meal, $tax, $tip) {
      $tax_amount = $meal * ($tax / 100);
      $tip_amount = $meal * ($tip / 100);
      return $meal + $tax_amount + $tip_amount;
  }
```

16. Web colors such as `#ffffff` and `#cc3399` are made by concatenating the hexadecimal color values for red, green, and blue. Write a function that accepts decimal red, green, and blue arguments and returns a string containing the appropriate color for use in a web page. For example, if the arguments are 255, 0, and 255, then the returned string should be `#ff00ff`. You may find it helpful to use the built-in function `dechex( )`, which is documented at http://www.php.net/dechex.