

Programming in PHP

X52.9224 Spring 2005

Instructor : David Mintz <dmintz@davidmintz.org>

http://davidmintz.org/php_course

Session 2

Outline

language basics continued
intro to PHP configuration
_GET and _POST data, and HTML form elements
functions, built-in and user-defined; variable scope

NOTE: Pieces of the following are stolen right out of the official PHP manual:

<http://php.net/manual/en/>

Alternative 'colon' syntax

PHP offers an alternative syntax for some of its control structures; namely, `if`, `while`, `for`, `foreach`, and `switch`. In each case, the basic form of the alternate syntax is to change the opening brace to a colon (`:`) and the closing brace to `endif;`, `endwhile;`, `endfor;`, `endforeach;`, or `endswitch;`, respectively.

```
<?php if ($a == 5): ?>
    A is equal to 5
<?php endif; ?>
```

The advantage is that it is more English-like, ergo more non-programmer friendly.

switch

The `switch` statement is similar to a series of `if` statements on the same expression. Sometimes you might want to compare the same variable (or expression) with many different values, and execute a different piece of code depending on which value it equals to. This is what the `switch` statement is for.

The following examples are two different ways to write the same thing, one using a series of `if` and `elseif` statements, and the other using the `switch` statement:

```
<?php
if ($i == 0) {
    echo "i equals 0";
} elseif ($i == 1) {
    echo "i equals 1";
} elseif ($i == 2) {
    echo "i equals 2";
```

```

}

switch ($i) {
case 0:
    echo "i equals 0";
    break;
case 1:
    echo "i equals 1";
    break;
case 2:
    echo "i equals 2";
    break;
}

```

It's important to understand that when PHP first starts stepping through the case statements, nothing happens unless and until it encounters a case value equal to the value in the switch statement. Once it does, it keeps executing all the statements until it encounters a break. Stated differently: you must explicitly break if you do not want execution to fall through to the next case.

A special case is the *default* case. It matches anything that wasn't matched by the other cases, and should be the last *case* statement.

```

<?php
switch ($i) {
case 0:
    echo "i equals 0";
    break;
case 1:
    echo "i equals 1";
    break;
case 2:
    echo "i equals 2";
    break;
default:
    echo "i is not equal to 0, 1 or 2";
}
?>

```

ternary operator

There is a shorthand way to say, basically, if/else.

```
echo $temperate > 80 ? "It's too hot" : "The temperate might be OK";
```

This means, if the condition before the ? is true, echo the next expression after it; otherwise, echo the thing after the colon. Some people like this construct, and some think it is evil. (Personally, I think it's OK if you don't abuse it by putting complicated expressions in it, or by nesting ternary constructs.)

Arrays

In Session One we said there were 8 data types in PHP. We covered the four main *scalar* data types: *boolean*, *float*, *integer*, and *string*. We skimmed over *NULL* and said we'd get into *resource* and *object* later in the course. We now move on to *arrays*.

You use arrays for storing more than one thing in the same variable, usually because those things are related to each other in some way. For example, as we will see shortly, PHP automatically provides two arrays for you that contain the data that is submitted to your script: `$_GET` and `$_POST`.

An array can be created by the `array()` language-construct. You can populate an array by passing the `array()` construct a number of comma-separated *key => value* pairs.

```
<?php
    $food_colors = array(
        'carrot' => 'orange' ,
        'apple'  => 'red'    ,
        'lemon'  => 'yellow' ,
        'spinach' => 'green'
    );
?>
```

A *key* may be either an *integer* or a *string*. *value* may be any value, including another array, object, or what have you. If a key is the standard representation of an integer, it will be interpreted as such (i.e. "8" will be interpreted as 8, while "08" will be interpreted as "08").

Note the commas separating the key/value pairs: they are obligatory *except* following the last key/value pair in the array, where it is unnecessary but harmless.

There are no different indexed and associative array types in PHP; there is only one array type, which can both contain integer and string indices.

You can create an array without specifying the keys:

```
<?php
    $foods = array('apple','banana' , 'beer' , 'braised bamboo fungus' );
?>
```

And if you do, PHP will provide integer keys for you automatically in numerical order starting from 0.

This might be a propitious time to jump ahead and introduce a PHP function called `print_r()`. It takes whatever variable you pass it (as *argument*) and prints out human-readable information about it, and it is handy for debugging. This example uses `print_r()` to demonstrate that `$foods` is indexed numerically:

```
<?php print_r($foods); ?>
```

Output:

```
Array
(
    [0] => apple
    [1] => banana
    [2] => beer
    [3] => braised bamboo fungus
)
```

In addition to the `array()` construct shown above, you can also set and get individual array elements explicitly using the following bracket notation:

```
<?php
    $food_colors['carrot'] = 'orange' ;
    $food_colors['apple'] = 'red' ;
    $food_colors['spinach'] = 'green' ;
    $food_colors['lemon'] = 'yellow' ;
    $food_colors['banana'] = 'yellow';
?>
```

If you assign a value to an array key that doesn't exist, it is automatically created; if it does exist, the corresponding value is overwritten. This behavior is because array keys by definition are unique (though the values need not be; you can have duplicates).

```
<?php
    echo "Well-ripened bananas are usually $food_colors[banana]";
    // guess the output
?>
```

You can add an element to the end of an array with empty bracket notation, like this:

```
<?php $foods[] = 'oil cured olives' ; ?>
```

You can use variables -- indeed any PHP expression -- in the brackets in place of a literal string or integer key, like so:

```
<?php

$my_favorite = 'banana';
echo "the color of my favorite food is $food_colors[$my_favorite]<br />";

// or ...

echo 'the last element in the $foods array is ' . $foods[ count($foods) - 1];

?>
```

(In the above example, as you may have inferred, `count()` is a function that takes an array and returns its size, i.e., the number of elements in the array. Since `$foods` is indexed numerically starting from 0, the index of the last element is equal to the size of the array minus one.)

A note about quoting array string/keys

We haven't discussed *constants* yet, so we will do so now.

You can define a constant by using the `define()` function. As the name suggests, once a constant is defined, it can never be changed or undefined. Only scalar data (boolean, integer, float and string) can be contained in constants. Though PHP doesn't require this, by convention, constant names are written in capitals.

```
<?php define ('WANTS_BEER' , true); ?>
```

You can get the value of a constant by simply specifying its name. Unlike with variables, you should not prepend a constant with a \$.

If you use an undefined constant, PHP assumes that you mean the name of the constant itself, just as if you called it as a string (CONSTANT vs "CONSTANT"). An error of level E_NOTICE will be issued when this happens. In sum, these are the differences between constants and variables:

- Constants do not have a dollar sign (\$) before them;
- Constants may only be defined using the `define()` function, not by simple assignment;
- Constants may be defined and accessed anywhere without regard to variable scoping rules (and we will get to scoping in short order);
- Constants may not be redefined or undefined once they have been set; and
- Constants may only evaluate to scalar values.

Getting back to arrays: If your array keys are strings, you should always quote them. I say "should" because PHP will let you get away with it but will emit a warning about an "undefined constant" – unless your string key happens to be the name of a defined constant. So, unless 'banana' happens to be the name of a constant,

```
<php echo $food_color[banana] ?>
```

is not a fatal error, but it definitely is to be avoided (why? e.g., if a future PHP version should happen introduce a predefined constant called `banana`, it will break your code). The exception is within double-quoted strings, where the above notation is perfectly OK, because PHP does not try to evaluate constants inside double-quoted strings.

Confusing? It may seem so, but it's all self-consistent. If you just use PHP and pay attention to what sets off warning notices, you will soon catch on.

foreach

When we discussed control structures in Session One, we said we would get to `foreach` after we had introduced arrays. So:

`foreach` provides a convenient way to *iterate* (or *loop*) through an array. If you're interested in just the values and not the keys:

```
<?php

$foods = ('sushi','french fries','carrots','broccoli');
echo "Today we have: <ul>" ;

foreach ( $foods as $food ) {
    echo "<li>$food</li>"
}
echo "</ul>" ;

/* exercise for the reader: get the same output
   using a for loop that has a count() call. hint:
   use a counter variable to index into the array.
*/
?>
```

If you are interested in the keys as well as the arrays, use this syntax:

```
<?php

$food_colors = array(

    'carrot' => 'orange' ,
    'apple'  => 'red'    ,
    'lemon'  => 'yellow' ,
    'spinach' => 'green' ,
    'banana' => 'yellow'
);

foreach ( $food_colors as $food => $color ) {

    echo "the food known as $food is normally $color<br />" ;
}
?>
```

PHP has a vast array (ouch, sorry) of built-in functions for doing practically anything you can imagine with arrays: sorting, searching, merging, computing the difference between arrays, computing the intersection between arrays, etc. For immediate gratification, go to <http://www.php.net/manual/en/ref.array.php>

the language construct `list`

You can assign an array to list of variables in one operation using the `list` language construct. `list` is useful in cases where the expression on the right evaluates to an array, but you would rather refer to its elements individually instead of using `$array[$x]` notation.

```
<?php
$name = "David";
$age = 46;
$favorite_beverage = "beer";
list($name, $age, $favorite_beverage) = $array;
?>
```

Note that this technique works only for numerically indexed arrays. Further, there's an assumption that you know the order of the elements in the array on the right side of the statement. For complete details please see <http://php.net/manual/en/function.list.php>

\$_GET and \$_POST and HTML forms

If your PHP script is accessed with GET parameters in the query string (appended to the URL following a ?), or if a form whose method is GET is submitted to your PHP page, then the input parameters are automatically available to your script in the \$_GET array.

If the input data is submitted in a POST request (typically, upon submission of a form whose method is POST), then that data is available to your script in an array called \$_POST.

\$_GET and \$_POST (among others; see <http://www.php.net/manual/en/language.variables.predefined.php>) are what's known as autoglobal arrays: PHP creates them for you and they are automatically available in every *scope*.

Assume a PHP called script.php. If I access it in my browser as

```
http://example.org/script.php?id=3&name=John
```

then script.php will be able to read \$_GET['id'] and see that its value is 3, and that \$_GET['name'] contains the value 'John'.

Assume a form whose method is POST, like so:

```
<form action="script.php" method="POST" '>
  Enter your name: <input type="text" name="name" />
  <input type="submit" value="Submit" />
</form>
```

Once the user clicks Submit to submit the form, the form variable named "name" is automatically available to script as \$_POST['name'].

It is easy and in many cases convenient to write PHP pages that submit to themselves. Thus script.php could be:

```
<?php
if (isset($_POST['name'])) {
    echo "Hello $_POST[name]!" ;
    exit;
} else {
    echo "Please introduce yourself. " ;
}
?>
<form action="script.php" method="POST" '>
  Enter your name: <input type="text" name="name" />
  <input type="submit" value="Submit" />
```

```
</form>
```

PHP provides another autoglobal array called `$_REQUEST` that contains everything: GET variables, POST variables, and cookies. If there happen to be variables that share the same name, one will overwrite the other, by default in this order: gpc (you can set this to something else with the `variables_order` directive in PHP's configuration file `php.ini`)

Let us look at how each of the HTML form elements can be handled. We will assume a POST request.

text input fields

We just saw a text field called 'name' in the above example. That's all there is to it.

HTML form submission: `<input type="text" name="some_text_input" />`
accessible to PHP as: `$_POST['some_text_input']`

textarea fields

These are handled just the same as text fields.

hidden fields

See above (-:

select (option) menus

With select menus whose 'multiple' attribute is not set, the selected item is the value that gets submitted.

HTML:
`<select name="color">`
 `<option name="green" >green</option>`
 `<option name="red" >red</option>`
 `<option name="blue" >blue</option>`
`</select>`

PHP: `$_POST['color']`

If the user selects 'red' and POSTs my form, the value of `$_POST['color']` will be 'red'.

If the SELECT menu's multiple attribute is set to true, then it's just a bit trickier, but you can handle it because PHP supports the submission of arrays. Use the bracket notation when you name your select form field.

```
<select name="colors[]" multiple="multiple" size="6">
  <option name="green" >green</option>
  <option name="red" >red</option>
  <option name="blue" >blue</option>
```

```

        <option name="white" >white</option>
        <option name="fuchsia" >fuchsia</option>
        <option name="gray" >gray</option>
</select>

```

When the user POSTs the form, there will be an element in the POST array called 'colors' which itself will be an array containing all the items from the menu that the user selected.

Note that if the user selects just one item, that value will still be in an array (rather than a string) within the POST array. Also, note that the name of the POSTed variable is going to be 'colors', rather than 'colors[]'.

```

<?php
if (!empty($_POST['colors'])) :
    echo "<pre>" ;
    print_r($_POST['colors']);
    echo "</pre>" ;

else:
    echo "please choose your favorite color(s)" ;
endif;
?>
<form action="<?php echo $_SERVER['PHP_SELF']?>" method="POST">
<select name="colors[]" multiple="multiple" size="6" >
    <option value="green" >green</option>
    <option value="red" >red</option>
    <option value="blue" >blue</option>
    <option value="white" >white</option>
    <option value="fuchsia" >fuchsia</option>
    <option value="gray" >gray</option>
</select>
<input type="submit" />
</form>

```

radio buttons

Working with radio buttons is the same as working with simple (non-multiple) select menus

HTML:

```

<input type="radio" name="cheese" value="Jarlsberg" /> Jarlsberg <br />
<input type="radio" name="cheese" value="Brie" /> Brie <br />
<input type="radio" name="cheese" value="Manchego" /> Manchego <br />

```

PHP: echo "I'm sorry sir, we're all out of \$_POST[cheese]" ;

checkboxes

If the checkbox is turned on, its value is submitted.

HTML:

```

<input type="checkbox" name="agreed" /> I agree to your outrageous terms

```

PHP:

```
if (! isset($_POST['agreed'] )) echo "Accept our terms or hit the road." ;
```

You can also make checkboxes behave like multiple select menus by using the bracket notation.

HTML:

Which of the following is your corporation engaged in? (Check all that apply)

```
<input type="checkbox" name="crimes[]" value="stealing" /> stealing <br />
<input type="checkbox" name="crimes[]" value="cheating" /> cheating <br />
<input type="checkbox" name="crimes[]" value="pillaging" />pillaging<br />
<input type="checkbox" name="crimes[]" value="looting" /> looting <br />
<input type="checkbox" name="crimes[]" value="plundering" /> plundering <br />
```

All the checked items will be submitted in the `$_POST['crimes']` array.

submit buttons

If the submit button has a name attribute, then the submit button's value will be submitted as a request variable named whatever the button's name attribute was – same as a textfield.

file uploads

The topic of file uploads merits its own discussion, and – you guessed it – we will deal with it later.

`$_SERVER`

Contains lots of information about the server environment, such as the name of the currently running PHP script, the server software, the user agent, etc.

```
<table>
<?php
foreach ($_SERVER as $key => $value) :

    ?><tr><td><?php echo $key ?></td><td><?php echo $value ?></td></tr>\n<?

endforeach;
?>
</table>
```

Not the least useful of these is `$_SERVER['PHP_SELF']`, which contains the name of the currently running script. You can use this variable as the action attribute of a FORM tag to make a page submit to itself, and avoid hard-coding the page's name.

Another handy one is `$_SERVER['REQUEST_METHOD']` which tells you if you're being GETed or POSTed.

register_globals

Once upon a time it was thought that it would be convenient to have a configuration directive `register_globals` which, if turned on, would automatically export all the `GET`, `POST`, `COOKIE`, and `SERVER` variables to the global scope. That means, for example, instead of `$_GET['id']` you could simply say `$id`. It proved harder to write secure PHP programs with `register_globals` on. Nowadays it is generally discouraged and `php.ini` ships with `register_globals` off by default.

Functions

You have already seen built-in PHP functions in action in some of the examples (because it is hard to do much without them). Functions are essential to code re-use and readability. There are two flavors of functions in PHP: those that are built in to your PHP interpreter, and those that you define yourself.

A function is a set of statements that can be executed just by invoking the function's name. Functions may be written to take *zero or more arguments* – data that the function receives as input.

A function typically does something with its input data, and often, *returns* some value to the caller. Otherwise the return type is said to be *void*; in fact if you assign to `$var` the return value of a function that does not explicitly return a non-NULL value, `$var` will be set to `NULL`. Remember that `NULL` is considered false; if you write a function that you want to return `TRUE` under certain circumstances, you have to do something affirmatively to make it happen (it isn't automatic).

This is worth repeating: the return value of a function that does not explicitly return a value is `NULL`.

And again, you can write functions that expect to be passed some data, or not; and you can write functions that return some data, or not.

Here is a trivial example to illustrate how we define functions.

```
<?php

function sum($x,$y) {

    $sum = $x + $y;
    return $sum;
}
?>
```

Note the keyword `function`, followed by the function name, followed by its *formal parameter list* (which can be empty), followed by a block of code in curly braces. This is called the function body.

The rather boring function above takes two parameters, `$x` and `$y`, and returns their sum.

Here is an example of how you would *call* this function:

```
<?php
    $x = 2;
    $y = 2;
    $sum = sum($x, $y); // function call
    echo " \$x + \$y is $sum<br />" ;
?>
```

On the third line of the above, where the function call occurs, the code in the function gets executed. When the function *returns*, the flow of execution returns to the caller. The `return` statement immediately ends execution of the current function, and returns its argument as the value of the function call.

Also, the arguments you pass to your function can be either literals, or variables, or more complex expressions.

```
<?php echo sum(24/8, 4+3); ?>
```

Functions do not have to explicitly return any value, nor do they have to contain a `return` statement. When a function finishes running, execution continues at the next statement following the function call.

```
<?php function hello() {
    echo "Hello everybody!";
}

echo "I am gonna call the hello() function<br />";
hello() ;
echo "<br />Now it's all over.<br />";
```

Note the parentheses in the function call: they are required, even if you are not passing any arguments to the function (because the parens tell PHP "this is a function").

Here is another function that might seem a little more useful. It takes an array as its argument and returns a string of HTML containing an unordered list whose items are the array elements.

```
<?php

function array2ul($array) {

    $retval = "<ul>\n<li>";
    $retval .= implode ("</li>\n<li>", $array);
```

```

        $retval .= "\n</ul>";
        return $retval;
    }

    $foods = array('apple','banana' , 'beer' , 'braised bamboo fungus' );
    echo array2ul($foods);

?>

```

This function leverages PHP's own `implode()` function, which takes an array and returns its elements collapsed into a string; the first argument is the "glue" that you want to use to separate the elements; the second is the array.

One of the points here is that you can chain function calls: a function can call another function which can call another function.

In fact, within a function body you can do practically anything you can do elsewhere, including declare nested functions. A function can also call itself. We call this recursion. Here's a version of `array2ul()` that supports multidimensional arrays:

```

<?php

function array2ul($array) {

    $retval = '<ul>';
    foreach ($array as $k => $v) {
        $retval .= "<li>$k: ";
        if (is_array($v)) {
            $retval .= array2ul($v);
        } else {
            $retval .= $v;
        }
        $retval .= '</li>';
    }
    $retval .= "</ul>";
    return $retval;
}

$foods = array(
    'apple',
    'banana' ,
    array(
        'peanuts',
        'coffee',
        array(
            'twinkies',
            'donuts'),
        'whiskey'),
    'beer' ,
    'braised bamboo fungus'
);

echo array2ul($foods);

?>

```

One thing you cannot do is redeclare a function – that is, declare a function by the same name as a function that has already been declared – nor can you undeclare a function once declared.

declaring default function parameters

If you declare a function to take one argument and call it with zero, PHP emits a warning notice – it thinks something is wrong, but it isn't fatal, so script execution continues. When you want some flexibility in this respect, you can declare default values for your function parameters:

```
<?php
function makecoffee($type = "cappuccino")
{
    return "Making a cup of $type.\n";
}
echo makecoffee();
echo makecoffee("espresso");
?>
```

Please see <http://www.php.net/manual/en/functions.arguments.php> for all the gory details and/or read the relevant sections of your favorite PHP book.

scope

The variables that you create inside a function are local to that function; the code outside that function cannot see them. Conversely, variables that are declared outside the function -- global variables -- are not visible to the function unless you make a little extra effort to access them. You can do that by using the `global` keyword to tell PHP you're talking about a global as (opposed to a local) variable. You can also access global variables by using the autoglobal `$GLOBALS` array.

```
<?php
function print_string() {

    print $string; // doesn't work; PHP considers this uninitialized

    global $string;
    print $string; // now it works
    print $GLOBALS['string']; // so does this

}

$string = "I am a global variable";
print_string();
?>
```

Reading a global from within a function is one thing, but in general, it is best to avoid *setting* global variables from within your functions, because if your script gets large and complex and different functions are doing that, you may lose your mind.

Handy Functions for Form Data Processing

PHP has something like 3,000 built-in functions. Certain of these are workhorses that you will have occasion to call on frequently when processing and displaying form input data.

isset()

isset() tells you whether the variable you pass it is set or not.

```
<?php
    if (! isset($_POST['name'])) {
        // they did not fill in the 'name' field
        echo "Please enter your name below" ;

    } else {

        echo "Hello " . htmlspecialchars( $_POST['name']);

    }
?>
```

empty()

Returns *false* if the variable whose name you pass has a non-empty and non-zero value. The distinction between

```
if (isset($var) )
```

and

```
if (!empty($var))
```

is rather subtle. The former will return true even if the value of \$var is false. The latter will return true if and only if \$var is set *and* has a true value. Don't worry if this is a bit confusing; we're just raising it now in the interest of disclosure (-: The point is that you will often be using one or the other to figure out what data has been submitted to your program (which you have to do in order to perform input validation).

htmlspecialchars()

Converts special characters to HTML entities. This function is useful in preventing user-supplied text from containing HTML markup, and you need it not only for ambitious applications like message boards, but also for safely re-displaying a form with the same values that the user provided (sometimes know as *sticky* form values).

```
<?php
if (isset($_POST['name' ])) {
    $name = htmlspecialchars($name) ;
} else {
```

```

        $name = '';
    }
    ?>
    <form action="<?php echo $_SERVER['PHP_SELF']?>" method="POST">
    Your name: <input type="text" value=" <?php echo $name ?>" />
    </form>

```

In this example, if the user submitted 'David "Killer" Mintz' in the name field and we did not escape the quotes, then the name field would be prematurely terminated when the browser encountered the first " following 'David '.

die()

Call `die()` to make your program do just that -- abort execution immediately. Sometimes there are circumstances where this is what you want. A synonym is `exit()`. If you can pass `die()` an optional argument, it will be printed out as your last gasp

```
<?php die("I can't take it any more"); ?>
```

or perhaps, when you do Assignment 2, to close your html document cleanly:

```
<?php exit("</body></html>") ; ?>
```

Post-installation: tweaking php.ini

When you did last week's assignment you probably noticed a lot of output when you clicked the link to display verbose information. That was the output of the `phpinfo()` function, which displays just about every imaginable detail about your PHP configuration and the environment. Writing and running a one-line script that simply says

```
<?php phpinfo(); ?>
```

will usually be the first thing you will do when testing a new PHP installation. One of the things it does it is read through your PHP configuration file – usually called `php.ini` – and report all the settings (directives), including those that govern the behavior of any optional extensions that you've installed. You can modify and tune PHP's behavior by re-setting these values and restarting your web server. (There are other ways of configuring PHP, which we will explore later.)

A few `php.ini` settings you should look at and possibly change:

Directive	Recommended value
<code>error_reporting</code>	<code>E_ALL</code>
<code>display_errors</code>	On
<code>magic_quotes_gpc</code>	Off

Directive	Recommended value
magic_quotes_runtime	Off
upload_tmp_dir	a directory that exists
register_globals	Off

The first item, *error_reporting*, tells PHP what level of verbosity to use in reporting errors. We are developing PHP and we want lots of debugging output. `E_ALL` is an integer constant that means max it out in versions below 5.0.0.

display_errors controls whether error output is sent to the browser (you can also write error messages to a log file, so *error_reporting* can be on even when *display_errors* is off.) We would like to see error output while testing our PHP pages, so we turn this on.

The *magic_quotes_** directives have to do with automatically escaping quotes inside strings so that you can safely use them in database queries, but it is more of an annoyance than anything else, so we turn it off.

upload_tmp_dir refers to where to store uploaded files temporarily before you copy them to wherever you wish to store them. It should be set to a directory that exists in the filesystem where PHP itself lives. The default is the system temp directory; but I have have uploads fail mysteriously when this was unset, and setting it solved the problem.

As mentioned earlier, *register_globals*, if enabled, exports all the request variables to the global scope. It's harder to code securely with *register_globals* than without. Nowadays everyone recommends turning it off.

There is a plethora of other directives, some of which we will discuss later. We should also point out that the optimal settings for development are different from those of a production environment. In the latter, best practice is to suppress error output for reasons of security and style.

Using the PHP Manual, Getting Help

The most precious resource for PHP programmers is <http://www.php.net/manual/en/> (or perhaps its translation into some language other than English). Please head over there and look around. When you're coding and learning PHP, you will sometimes wonder whether PHP supplies a function that will help you do something, and you will often find that it does. Look at the function reference and notice how they document the functions with a one-liner that describes its behavior; information about what PHP versions support it; a function prototype showing the data types of the arguments and return value; usage examples followed by discussion; and user-contributed notes.

If RTFM doesn't help, there are forums, mail lists, code snippet libraries, etc., too numerous to mention. (If this goes without saying, don't take offense, but: please be a good citizen and do

your best to exhaust all the self-help resources you can find before posting a request someplace.) A list of PHP sites is at <http://www.php.net/links.php>, and other support resources are listed at <http://www.php.net/support.php>. One of the world's finest PHP user groups happens to be based right in New York City: <http://nyphp.org/>. They have (free) monthly meetings featuring outstanding speakers on advanced topics.

Assignment

informal/warm up exercises

These are just for your benefit; please do not hand them in. Just write the code and test it to make sure it works.

Create a numerically indexed array and populate it with some data. Then iterate through it with a `for` loop, printing each element.

Create a so-called associative array (one with string keys) and populate it. Write a `foreach` loop to iterate through it printing keys and values.

Create an array of integers that are numeric, but don't insert them in numeric order. Then go to the PHP online documentation, or consult a book, and find out how to sort the array in ascending numeric order.

Write a function that takes an associative array as input, and prints an HTML table with two columns: the keys on the left and the values on the right.

Write a function that simply returns the string "hello world" to the caller.

Now rewrite it to take a string argument `$name` so that it returns "hello `$name`".

Now rewrite it so that it takes a string argument whose default value is the string "you." Thus if I call it with no arguments, it returns "hello you." If I call it with the string "Bart" it returns "hello Bart."

Homework

Write a program called `week2.php` that displays a form that submits to itself.

When initially accessed, `week2.php` displays an HTML form. If the user POSTs the form, `week2.php` will simply output the names and values (if any) of each of the form elements, and exit. Use `htmlspecialchars()` on the form data that you display. If one of your form fields is unset (i.e., the checkbox), output something like *[no value]* in place of the value.

Copy the following from the class website and use it for your assignment, replacing the just the multi-line comment with the necessary PHP code.

```
/* week2.php */
<?php echo '<?xml version="1.0" encoding="iso-8859-1"?>'><!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>Programming PHP: Session Two</title>
    <style type="text/css">
      body {
        font-family : verdana, arial, helvetica, sans-serif;
        font-size : small;
        line-height : 120%;
        margin-left: 20%;
      }
      form, textarea {
        font-family : verdana, arial, helvetica, sans-serif;
        font-size : small;
      }
    </style>
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
  </head>
  <body style="margin-top:80px">
  <?php

/*

Replace this comment with PHP that does the following:

if the form is POSTed, display the names and values of all the form elements
(whether set or not), using htmlspecialchars() on all of the values. then
exit.

otherwise, just display the form (i.e., do nothing and let execution continue).

*/

?>

<form action="<?php echo $_SERVER['PHP_SELF']?>" method="post">
<table>
<tr valign="middle">
  <td align="right">
    Your name
  </td>
  <td>
    <input name="your_name" type="text" size="20" />
  </td>
</tr>
<tr valign="top">
  <td align="right">
    Comments
  </td>
  <td>
    <textarea name="comments" rows="3" cols="40"></textarea>
  </td>
</tr>
<tr valign="middle">
  <td align="right">
    Gender
  </td>
  <td>
```

