Chapter 2. Working with Text and Numbers

PHP can work with different types of data. In this chapter, you'll learn about individual values such as numbers and single pieces of text. You'll learn how to put text and numbers in your programs, as well as some of the limitations the PHP interpreter puts on those values and some common tricks for manipulating them.

Most PHP programs spend a lot of time handling text because they spend a lot of time generating HTML and working with information in a database. HTML is just a specially formatted kind of text, and information in a database, such as a username, a product description, or an address is a piece of text, too. Slicing and dicing text easily means you can build dynamic web pages easily.

In Chapter 1, you saw variables in action, but this chapter teaches you more about them. A variable is a named container that holds a value. The value that a variable holds can change as a program runs. When you access data submitted from a form or exchange data with a database, you use variables. In real life, a variable is something such as your checking account balance. As time goes on, the value that the phrase "checking account balance" refers to fluctuates. In a PHP program, a variable might hold the value of a submitted form parameter. Each time the program runs, the value of the submitted form parameter can be different. But whatever the value, you can always refer to it by the same name. This chapter also explains in more detail what variables are: how you create them and do things such as change their values or print them.

2.1 Text

When they're used in computer programs, pieces of text are called *strings*. This is because they consist of individual characters, strung together. Strings can contain letters, numbers, punctuation, spaces, tabs, or any other characters. Some examples of strings are `I would like 1 bowl of soup`, and `"Is it too hot?" he asked`, and `There's no spoon!`. A string can even contain the contents of a binary file such as an image or a sound. The only limit to the length of a string in a PHP program is the amount of memory your computer has.

2.1.1 Defining Text Strings

There are a few ways to indicate a string in a PHP program. The simplest is to surround the string with single quotes:

```
print 'I would like a bowl of soup.';
print 'chicken';
print '06520';
print '"I am eating dinner," he growled.';
```

Since the string consists of everything inside the single quotes, that's what is printed:

```
I would like a bowl of soup.chicken06520"I am eating dinner," he growled.
```

The output of those four `print` statements appears all on one line. No linebreaks are added by `print`.[1]

[1] You may also see `echo` used in some PHP programs to print text. It works just like `print`.

The single quotes aren't part of the string. They are *delimiters*, which tell the PHP interpreter where the start and end of the string is. If you want to include a single quote inside a string surrounded with single quotes, put a backslash (`\`) before the single quote inside the string:

```
print 'We\'ll each have a bowl of soup.';
```

The `\'` sequence is turned into ' inside the string, so what is printed is:

```
We'll each have a bowl of soup.
```

The backslash tells the PHP interpreter to treat the following character as a literal single quote instead of the single quote that means "end of string." This is called *escaping*, and the backslash is called the *escape character*. An escape character tells the system to do something special with the character that comes after it. Inside a single-quoted string, a single quote usually means "end of string." Preceding the single quote with a backslash changes its meaning to a literal single quote character.

Curly Quotes and Text Editors

Word processors often automatically turn straight quotes like ' and " into curly quotes like ‘, ’, “, and ”. The PHP interpreter only understands straight quotes as string delimiters. If you're writing PHP programs in a word processor or text editor that puts curly quotes in your programs, you have two choices: tell your word processor to stop it or use a different one. A program such as emacs, vi, BBEdit, or Windows Notepad leaves your quotes alone.

The escape character can itself be escaped. To include a literal backslash character in a string, put a back slash before it:

```
print 'Use a \\ to escape in a string';
```

This prints:

```
Use a \ to escape in a string
```

The first backslash is the escape character: it tells the PHP interpreter that something different is going on with the next character. This affects the second backslash: instead of the special action ("treat the next character literally"), a literal backslash is included in the string.

Note that these are backslashes that go from top left to bottom right, not forward slashes that go from bottom left to top right. Remember that two forward slashes (`//`) indicate a comment.

You can include whitespace such as newlines in single-quoted strings:

```
print '<ul>
<li>Beef Chow-Fun</li>
<li>Sauteed Pea Shoots</li>
<li>Soy Sauce Noodles</li>
</ul>';
```

This puts the HTML on multiple lines:

```
<ul>
<li>Beef Chow-Fun</li>
<li>Sauteed Pea Shoots</li>
<li>Soy Sauce Noodles</li>
</ul>
```

Since the single quote that marks the end of the string is immediately after the `</ul>`, there is no newline at the end of the string.

The only characters that get special treatment inside single-quoted strings are backslash and single quote. Everything else is treated literally.

You can also delimit strings with double quotes. Double-quoted strings are similar to single-quoted strings, but they have more special characters. These special characters are listed in Table 2-1.

Table 2-1. Special characters in double-quoted strings

Character	Meaning
\n	Newline (ASCII 10)
\r	Carriage return (ASCII 13)
\t	Tab (ASCII 9)
\\	\
\$	$
\"	"
\0 .. \777	Octal (base 8) number
\x0 .. \xFF	Hexadecimal (base 16) number

The biggest difference between single-quoted and double-quoted strings is that when you include variable names inside a double-quoted string, the value of the variable is substituted into the string, which doesn't happen with single-quoted

strings. For example, if the variable `$user` held the value `Bill`, then '`Hi $user`' is just that: `Hi $user`. However, "`Hi $user`" is `Hi Bill`. I get into this in more detail later in this chapter in Section 2.3.

As mentioned in Section 1.3, you can also define strings with the *here document* syntax. A here document begins with `<<<` and a delimiter word. It ends with the same word at the beginning of a line. Example 2-1 shows a here document.

Example 2-1. Here document

```
<<<HTMLBLOCK
<html>
<head><title>Menu</title></head>
<body bgcolor="#fffed9">
<h1>Dinner</h1>
<ul>
  <li> Beef Chow-Fun
  <li> Sauteed Pea Shoots
  <li> Soy Sauce Noodles
  </ul>
</body>
</html>
HTMLBLOCK
```

In Example 2-1, the delimiter word is `HTMLBLOCK`. Here document delimiters can contain letters, numbers, and the underscore character. The first character of the delimiter must be a letter or the underscore. It's a good idea to make all the letters in your here document delimiters uppercase to visually set off the here document. The delimiter that ends the here document must be alone on its line. The delimiter can't be indented and no whitespace, comments, or other characters are allowed after it. The only exception to this is that a semicolon is allowed immediately after the delimiter to end a statement. In that case, nothing can be on the same line after the semicolon. The code in Example 2-2 follows these rules to print a here document.

Example 2-2. Printing a here document

```
print <<<HTMLBLOCK
<html>
<head><title>Menu</title></head>
<body bgcolor="#fffed9">
<h1>Dinner</h1>
<ul>
  <li> Beef Chow-Fun
  <li> Sauteed Pea Shoots
  <li> Soy Sauce Noodles
  </ul>
</body>
</html>
HTMLBLOCK;
```

Here documents obey the same escape-character and variable substitution rules as double-quoted strings. These make them especially useful when you want to define or print a string that contains a lot of text or HTML with some variables mixed in. Later on in the chapter, Example 2-22 demonstrates this.

To combine two strings, use a . (period), the string concatenation operator. Here are some combined strings:

```
print 'bread' . 'fruit';
print "It's a beautiful day " . 'in the neighborhood.';
print "The price is: " . '$3.95';
print 'Inky' . 'Pinky' . 'Blinky' . 'Clyde';
```

The combined strings print as:

```
breadfruit
It's a beautiful day in the neighborhood.
The price is: $3.95
InkyPinkyBlinkyClyde
```

2.1.2 Manipulating Text

PHP has a number of built-in functions that are useful when working with strings. This section introduces the functions that are most helpful for two common tasks: validation and formatting. The "Strings" chapter of the PHP online manual, at http://www.php.net/strings, has information on other built-in string handling functions.

2.1.2.1 Validating strings

Validation is the process of checking that input coming from an external source conforms to an expected format or meaning. It's making sure that a user really entered a ZIP Code in the "ZIP Code" box of a form or a reasonable email address in the appropriate place. Chapter 6 delves into all the aspects of form handling, but since submitted form data is provided to your PHP programs as strings, this section discusses how to validate those strings.

The `trim( )` function removes whitespace from the beginning and end of a string. Combined with `strlen( )`, which tells you the length of a string, you can find out the length of a submitted value while ignoring any leading or trailing spaces. Example 2-3 shows you how. (Chapter 3 discusses in more detail the `if( )` statement used in Example 2-3.)

Example 2-3. Checking the length of a trimmed string

```
// $_POST['zipcode'] holds the value of the submitted form parameter
// "zipcode"
$zipcode = trim($_POST['zipcode']);
// Now $zipcode holds that value, with any leading or trailing spaces
// removed
$zip_length = strlen($zipcode);
// Complain if the ZIP code is not 5 characters long
if ($zip_length != 5) {
    print "Please enter a ZIP code that is 5 characters long.";
}
```

Using `trim( )` protects against someone who types a ZIP Code of `732` followed by two spaces. Sometimes the extra spaces are accidental and sometimes they are malicious. Whatever the reason, throw them away when appropriate to make sure that you're getting the string length you care about.

You can chain together the calls to `trim( )` and `strlen( )` for more concise code. Example 2-4 does the same thing as Example 2-3.

Example 2-4. Concisely checking the length of a trimmed string

```
if (strlen(trim($_POST['zipcode'])) != 5) {
    print "Please enter a ZIP code that is 5 characters long.";
}
```

Four things happen in the first line of Example 2-4. First, the value of the variable `$_POST['zipcode']` is passed to the `trim( )` function. Second, the return value of that function — `$_POST['zipcode']` with leading and trailing whitespace removed — is handed off to the `strlen( )` function, which then returns the length of the trimmed string. Third, this length is compared with 5. Last, if the length is not equal to 5, then the `print` statement inside the `if( )` block runs.

To compare two strings, use the equality operator (`= =`), as shown in Example 2-5.

Example 2-5. Comparing strings with the equality operator

```
if ($_POST['email'] == 'president@whitehouse.gov') {
   print "Welcome, Mr. President.";
}
```

The `print` statement in Example 2-5 runs only if the submitted form parameter `email` is the all-lowercase `president@whitehouse.gov`. When you compare strings with `= =`, case is important. `president@whitehouse.GOV` is not the same as `President@Whitehouse.Gov` or `president@whitehouse.gov`.

To compare strings without paying attention to case, use `strcasecmp( )`. It compares two strings while ignoring differences in capitalization. If the two strings you provide to `strcasecmp( )` are the same (independent of any differences between upper- and lowercase letters), it returns 0. Example 2-6 shows how to use `strcasecmp( )`.

Example 2-6. Comparing strings case-insensitively

```
if (strcasecmp($_POST['email'], 'president@whitehouse.gov') == 0) {
    print "Welcome back, Mr. President.";
}
```

The `print` statement in Example 2-6 runs if the submitted form parameter `email` is `President@Whitehouse.Gov`, `PRESIDENT@WHITEHOUSE.GOV`, `presIDENT@whiteHOUSE.GoV`, or any other capitalization of `president@whitehouse.gov`.

2.1.2.2 Formatting text

The `printf( )` function gives you more control (compared to `print`) over how the output looks. You pass `printf( )` a format string and a bunch of items to print. Each rule in the format string is replaced by one item. Example 2-7 shows `printf( )` in action.

Example 2-7. Formatting a price with printf( )

```
$price = 5; $tax = 0.075;
```

```
printf('The dish costs $%.2f', $price * (1 + $tax));
```

This prints:

```
The dish costs $5.38
```

In Example 2-7, the format rule `%.2f` is replaced with the value of `$price * (1 + $tax)` and formatted so that it has two decimal places.

Format string rules begin with `%` and then have some optional modifiers that affect what the rule does:

A padding character

> If the string that is replacing the format rule is too short, this is used to pad it. Use a space to pad with spaces or a `0` to pad with zeroes.

A sign

> For numbers, a plus sign (+) makes `printf( )` put a + before positive numbers (normally, they're printed without a sign.) For strings, a minus sign (-) makes `printf( )` right justify the string (normally, they're left justified.)

A minimum width

> The minimum size that the value replacing the format rule should be. If it's shorter, then the padding character is used to beef it up.

A period and a precision number

> For floating-point numbers, this controls how many digits go after the decimal point. In Example 2-7, this is the only modifier present. The `.2` formats `$price + (1 + $tax)` with two decimal places.

After the modifiers come a mandatory character that indicates what kind of value should be printed. The three discussed here are `d` for decimal number, `s` for string, and `f` for floating-point number.

If this stew of percent signs and modifiers has you scratching your head, don't worry. The most frequent use of `printf( )` is probably to format prices with the `%.2f` format rule as shown in Example 2-7. If you absorb nothing else about `printf( )` for now, just remember that it's your go-to function when you want to format a decimal value.

But if you delve a little deeper, you can do some other handy things with it. For example, using the `0` padding character and a minimum width, you can format a date or ZIP Code properly with leading zeroes, as shown in Example 2-8.

Example 2-8. Zero-padding with printf( )

```
$zip = '6520';
$month = 2;
$day = 6;
$year = 2007;

printf("ZIP is %05d and the date is %02d/%02d/%d", $zip, $month, $day, $year);
```

Example 2-8 prints:

```
ZIP is 06520 and the date is 02/06/2007
```

The sign modifier is helpful for explicitly indicating positive and negative values. Example 2-9 uses it to display a some temperatures.

Example 2-9. Displaying signs with printf( )

```
$min = -40;
$max = 40;
printf("The computer can operate between %+d and %+d degrees Celsius.", $min, $max);
```

Example 2-9 prints:

```
The computer can operate between -40 and +40 degrees Celsius.
```

To learn about other `printf( )` format rules, visit http://www.php.net/sprintf.

Another kind of text formatting is to manipulate the case of strings. The `strtolower( )` and `strtoupper( )` functions make all-lowercase and all-uppercase versions, respectively, of a string. Example 2-10 shows `strtolower( )` and `strtoupper( )` at work.

Example 2-10. Changing case

```
print strtolower('Beef, CHICKEN, Pork, duCK');
print strtoupper('Beef, CHICKEN, Pork, duCK');
```

Example 2-10 prints:

```
beef, chicken, pork, duck
BEEF, CHICKEN, PORK, DUCK
```

The `ucwords( )` function uppercases the first letter of each word in a string. This is useful when combined with `strtolower( )` to produce nicely capitalized names when they are provided to you in all uppercase. Example 2-11 shows how to combine `strtolower( )` and `ucwords( )`.

Example 2-11. Prettifying names with ucwords( )

```
print ucwords(strtolower('JOHN FRANKENHEIMER'));
```

Example 2-11 prints:

```
John Frankenheimer
```

With the `substr( )` function, you can extract just part of a string. For example, you may only want to display the beginnings of messages on a summary page. Example 2-12 shows how to use `substr( )` to truncate the submitted form parameter `comments`.

Example 2-12. Truncating a string with substr( )

```
// Grab the first thirty characters of $_POST['comments']
print substr($_POST['comments'], 0, 30);
// Add an ellipsis
print '...';
```

If the submitted form parameter `comments` is:

```
The Fresh Fish with Rice Noodle was delicious, but I didn't like the Beef Tripe.
```

Example 2-12 prints:

```
The Fresh Fish with Rice Noodl...
```

The three arguments to `substr( )` are the string to work with, the starting position of the substring to extract, and the number of characters to extract. The beginning of the string is position 0, not 1, so `substr($_POST['comments'], 0, 30)` means "extract 30 characters from `$_POST['comments']` starting at the beginning of the string."

When you give `substr( )` a negative number for a start position, it counts back from the end of the string to figure out where to start. A start position of -4 means "start four characters from the end." Example 2-13 uses a negative start position to display just the last four digits of a credit card number.

Example 2-13. Extracting the end of a string with substr( )

```
print 'Card: XX';
print substr($_POST['card'],-4,4);
```

If the submitted form parameter `card` is `4000-1234-5678-9101`, Example 2-13 prints:

```
Card: XX9101
```

As a shortcut, use `substr($_POST['card'],-4)` instead of `substr($_POST['card'], -4,4)`. When you leave out the last argument, `substr( )` returns everything from the starting position (whether positive or negative) to the end of the string.

Instead of extracting a substring, the `str_replace( )` function changes parts of a string. It looks for a substring and replaces the substring with a new string. This is useful for simple template-based customization of HTML. Example 2-14 uses `str_replace( )` to set the `class` attribute of `<span>` tags.

Example 2-14. Using str_replace( )

```
print str_replace('{class}',$my_class,
                  '<span class="{class}">Fried Bean Curd<span>
                   <span class="{class}">Oil-Soaked Fish</span>');
```

If `$my_class` is `lunch`, then Example 2-14 prints:

```
<span class="lunch">Fried Bean Curd<span>
<span class="lunch">Oil-Soaked Fish</span>
```

Each instance of `{class}` (the first argument to `str_replace( )`) is replaced by `lunch` (the value of `$my_class`) in the string that is the third argument passed to `str_replace( )`.

2.2 Numbers

Numbers in PHP are expressed using familiar notation, although you can't use commas or any other characters to group thousands. You don't have to do anything special to use a number with a decimal part as compared to an integer. Example 2-15 lists some valid numbers in PHP.

Example 2-15. Numbers

```
print 56;
print 56.3;
print 56.30;
print 0.774422;
print 16777.216;
print 0;
print -213;
print 1298317;
print -9912111;
print -12.52222;
print 0.00;
```

2.2.1 Using Different Kinds of Numbers

Internally, the PHP interpreter makes a distinction between numbers with a decimal part and those without one. The former are called *floating-point* numbers and the latter are called *integers*. Floating-point numbers take their name from the fact that the decimal point can "float" around to represent different amounts of precision.

The PHP interpreter uses the math facilities of your operating system to represent numbers so the largest and smallest numbers you can use, as well as the number of decimal places you can have in a floating-point number, vary on different systems.

One distinction between the PHP interpreter's internal representation of integers and floating-point numbers is the exactness of how they're stored. The integer 47 is stored as exactly 47. The floating-point number 46.3 could be stored as 46.2999999. This affects the correct technique of how to compare numbers. Section 3.3 explains comparisons and shows how to properly compare floating-point numbers.

2.2.2 Arithmetic Operators

Doing math in PHP is a lot like doing math in elementary school, except it's much faster. Some basic operations between numbers are shown in Example 2-16.

Example 2-16. Math operations

```
print 2 + 2;
print 17 - 3.5;
print 10 / 3;
print 6 * 9;
```

The output of Example 2-16 is:

```
4
13.5
3.3333333333333
54
```

In addition to the plus sign (`+`) for addition, the minus sign (`-`) for subtraction, the forward slash (`/`) for division, and the asterisk (`*`) for multiplication, PHP also supports the percent sign (`%`) for modulus division. This returns the remainder of a division operation:

```
print 17 % 3;
```

This prints:

```
2
```

Since 17 divided by 3 is 5 with a remainder of 2, 17 % 3 equals 2. The modulus operator is most useful for printing rows whose colors alternate in an HTML table, as shown in Example 4-12.

The arithmetic operators, as well as the other PHP operators that you'll meet later in the book, fit into a strict precedence of operations. This is how the PHP interpreter decides in what order to do calculations if they are written ambiguously. For example, "3 + 4 * 2" could mean "add 3 and 4 and then multiply the result by 2," which results in 14. Or, it could mean "add 3 to the product of 4 and 2," which results in 11. In PHP (as well as the math world in general), multiplication has a

higher precedence than addition, so the second interpretation is correct. First, the PHP interpreter multiplies 4 and 2, and then it adds 3 to the result.

The precedence table of all PHP operators is part of the online PHP Manual at http://www.php.net/language.operators#language.operators.precedence. You can avoid memorizing or repeatedly referring to this table, however, with a healthy dose of parentheses. Grouping operations inside parentheses unambiguously tells the PHP interpreter to do what's inside the parentheses first. The expression "(3 + 4) * 2" means "add 3 and 4 and then multiply the result by 2." The expression "3 + (4 * 2)" means "multiply 4 and 2 and then add 3 to the result."

Like other modern programming languages, you don't have to do anything special to ensure that the results of your calculations are properly represented as integers or floating-point numbers. Dividing one integer by another produces a floating-point result if the two integers don't divide evenly. Similarly, if you do something to an integer that makes it larger than the maximum allowable integer or smaller than the minimum possible integer, the PHP interpreter converts the result into a floating-point number so you get the proper result for your calculation.

2.3 Variables

Variables hold the data that your program manipulates while it runs, such as information about a user that you've loaded from a database or entries that have been typed into an HTML form. In PHP, variables are denoted by `$` followed by the variable's name. To assign a value to a variable, use an equals sign (`=`). This is known as the assignment operator.

```
$plates = 5;
$dinner = 'Beef Chow-Fun';
$cost_of_dinner = 8.95;
$cost_of_lunch = $cost_of_dinner;
```

Assignment works with here documents as well:

```
$page_header = <<<HTML_HEADER
<html>
<head><title>Menu</title></head>
<body bgcolor="#fffed9">
<h1>Dinner</h1>
HTML_HEADER;

$page_footer = <<<HTML_FOOTER
</body>
</html>
HTML_FOOTER;
```

Variable names must begin with letter or an underscore. The rest of the characters in the variable name may be letters, numbers, or an underscore. Table 2-2 lists some acceptable variable names.

Table 2-2. Acceptable variable names

Acceptable
`$size`
`$drinkSize`
`$my_drink_size`
`$_drinks`
`$drink4you2`

Table 2-3 lists some unacceptable variable names and what's wrong with them.

Table 2-3. Unacceptable variable names

Variable name	Flaw
`$2hot4u`	Begins with a number
`$drink-size`	Unacceptable character: `-`
`$drinkmaster@example.com`	Unacceptable characters: `@` and .
`$drink!lots`	Unacceptable character: `!`
`$drink+dinner`	Unacceptable character: `+`

Variable names are case-sensitive. This means that variables named `$dinner`, `$Dinner`, and `$DINNER` are separate and distinct, with no more in common than if they were named `$breakfast`, `$lunch`, and `$supper`. In practice, you should avoid using variable names that differ only by letter case. They make programs difficult to read and debug.

2.3.1 Operating on Variables

Arithmetic and string operators work on variables containing numbers or strings just like they do on literal numbers or strings. Example 2-17 shows some math and string operations at work on variables.

Example 2-17. Operating on variables

```
<?php
$price = 3.95;
$tax_rate = 0.08;
$tax_amount = $price * $tax_rate;
$total_cost = $price + $tax_amount;

$username = 'james';
$domain = '@example.com';
$email_address = $username . $domain;

print 'The tax is ' . $tax_amount;
print "\n"; // this prints a linebreak
```

```
print 'The total cost is ' .$total_cost;
print "\n"; // this prints a linebreak
print $email_address;
?>
```

Example 2-17 prints:

```
The tax is 0.316
The total cost is 4.266
james@example.com
```

The assignment operator can be combined with arithmetic and string operators for a concise way to modify a value. An operator followed by the equals sign means "apply this operator to the variable." Example 2-18 shows two identical ways to add 3 to `$price`.

Example 2-18. Combined assignment and addition

```
// Add 3 the regular way
$price = $price + 3;
// Add 3 with the combined operator
$price += 3;
```

Combining the assignment operator with the string concatenation operator appends a value to a string. Example 2-19 shows two identical ways to add a suffix to a string. The advantage of the combined operators is that they are more concise.

Example 2-19. Combined assignment and concatenation

```
$username = 'james';
$domain = '@example.com';

// Concatenate $domain to the end of $username the regular way
$username = $username . $domain;
// Concatenate with the combined operator
$username .= $domain;
```

Incrementing and decrementing variables by 1 are so common that these operations have their own operators. The `++` operator adds 1 to a variable, and the `--` operator subtracts 1. These operators are usually used in `for( )` loops, which are detailed in Chapter 3. But you can use them on any variable holding a number, as shown in Example 2-20.

Example 2-20. Incrementing and decrementing

```
// Add one to $birthday
$birthday = $birthday + 1;
// Add another one to $birthday
++$birthday;

// Subtract 1 from $years_left
$years_left = $years_left - 1;
// Subtract another 1 from $years_left
```

```
--$years_left;
```

2.3.2 Putting Variables Inside Strings

Frequently, you print the values of variables combined with other text, such as when you display an HTML table with calculated values in the cells or a user profile page that shows a particular user's information in a standardized HTML template. Double-quoted strings and here documents have a property that makes this easy: you can *interpolate* variables into them. This means that if the string contains a variable name, the variable name is replaced by the value of the variable. In Example 2-21, the value of `$email` is interpolated into the printed string.

Example 2-21. Variable interpolation

```
$email = 'jacob@example.com';
print "Send replies to: $email";
```

Example 2-21 prints:

```
Send replies to: jacob@example.com
```

Here documents are especially useful for interpolating many variables into a long block of HTML, as shown in Example 2-22.

Example 2-22. Interpolating in a here document

```
$page_title = 'Menu';
$meat = 'pork';
$vegetable = 'bean sprout';
print <<<MENU
<html>
<head><title>$page_title</title></head>
<body>
<ul>
<li> Barbecued $meat
<li> Sliced $meat
<li> Braised $meat with $vegetable
</ul>
</body>
</html>
MENU;
```

Example 2-22 prints:

```
<html>
<head><title>Menu</title></head>
<body>
<ul>
<li> Barbecued pork
<li> Sliced pork
<li> Braised pork with bean sprout
```

```
</ul>
</body>
```

When you interpolate a variable into a string in a place where the PHP interpreter could be confused about the variable name, surround the variable with curly braces to remove the confusion. Example 2-23 needs curly braces so that `$preparation` is interpolated properly.

Example 2-23. Interpolating with curly braces

```
$preparation = 'Braise';
$meat = 'Beef';
print "{$preparation}d $meat with Vegetables";
```

Example 2-23 prints:

```
Braised Beef with Vegetables
```

Without the curly braces, the print statement in Example 2-23 would be `print "$preparationd $meat with Vegetables";`. In that statement, it looks like the variable to interpolate is named `$preparationd`. The curly braces are necessary to indicate where the variable name stops and the literal string begins. The curly brace syntax is also useful for interpolating more complicated expressions and array values, discussed in Chapter 4.

2.4 Chapter Summary

Chapter 2 covers:

- Defining strings in your programs three different ways: with single quotes, with double quotes, and as a here document.
- Escaping: what it is and what characters need to be escaped in each kind of string.
- Validating a string by checking its length, removing leading and trailing whitespace from it, or comparing it to another string.
- Formatting a string with `printf( )`.
- Manipulating the case of a string with `strtolower( )`, `strtoupper( )`, or `ucwords( )`.
- Selecting part of a string with `substr( )`.
- Changing part of a string with `str_replace( )`.
- Defining numbers in your programs.
- Doing math with numbers.
- Storing values in variables.
- Naming variables appropriately.
- Using combined operators with variables.
- Using increment and decrement operators with variables.
- Interpolating variables in strings.

2.5 Exercises

1. Find the errors in this PHP program:

```
  <? php
  print 'How are you?';
  print 'I'm fine.';
??>
```

5. Write a PHP program that computes the total cost of this restaurant meal: two hamburgers at $4.95 each, one chocolate milk shake at $1.95, and one cola at 85 cents. The sales tax rate is 7.5%, and you left a pre-tax tip of 16%.
6. Modify your solution to the previous exercise to print out a formatted bill. For each item in the meal, print the price, quantity, and total cost. Print the pre-tax food and drink total, the post-tax total, and the total with tax and tip. Make sure that prices in your output are vertically aligned.
7. Write a PHP program that sets the variable `$first_name` to your first name and `$last_name` to your last name. Print out a string containing your first and last name separated by a space. Also print out the length of that string.
8. Write a PHP program that uses the increment operator (`++`) and the combined multiplication operator (`*=`) to print out the numbers from 1 to 5 and powers of 2 from 2 (2^1) to 32 (2^5).
9. Add comments to the PHP programs you've written for the other exercises. Try both single and multiline comments. After you've added the comments, run the programs to make sure they work properly and your comment syntax is correct.