Chapter 3. Making Decisions and Repeating Yourself

Chapter 2 covered the basics of how to represent data in PHP programs. A program full of data is only half complete, though. The other piece of the puzzle is using that data to control how the program runs, taking actions such as:

- If an administrative user is logged in, print a special menu.
- Print a different page header if it's after three o'clock.
- Notify a user if new messages have been posted since she last logged in.

All of these actions have something in common: they make decisions about whether a certain logical condition involving data is true or false. In the first action, the logical condition is "Is an administrative user logged in?" If the condition is true (yes, an administrative user is logged in), then a special menu is printed. The same kind of thing happens in the next example. If the condition "is it after three o'clock?" is true, then a different page header is printed. Likewise, if "Have new messages been posted since the user last logged in?" is true, then the user is notified.

When making decisions, the PHP interpreter boils down an expression into `true` or `false`. Section 3.1 explains how the interpreter decides which expressions and values are `true` and which are `false`.

Those `true` and `false` values are used by language constructs such as `if( )` to decide whether to run certain statements in a program. The ins and outs of `if( )` are detailed later in this chapter in Section 3.2. Use `if( )` and similar constructs any time the outcome of a program depends on some changing conditions.

While `true` and `false` are the cornerstones of decision making, usually you want to ask more complicated questions, such as "is this user at least 21 years old?" or "does this user have a monthly subscription to the web site or enough money in their account to buy a daily pass?" Section 3.3, later in this chapter, explains PHP's comparison and logical operators. These help you express whatever kind of decision you need to make in a program, such as seeing whether numbers or strings are greater than or less than each other. You can also chain together decisions into a larger decision that depends on its pieces.

Decision making is also used in programs when you want to repeatedly execute certain statements — you need a way to indicate when the repetition should stop. Frequently, this is determined by a simple counter, such as "repeat 10 times." This is like asking the question "Have I repeated 10 times yet?" If so, then the program continues. If not, the action is repeated again. Determining when to stop can be more complicated, too — for example, "show another math question to a student until 6 questions have been answered correctly." Section 3.4, later in this chapter, introduces PHP's `while( )` and `for( )` constructs, with which you can implement these kinds of loops.

3.1 Understanding true and false

Every expression in a PHP program has a truth value: `true` or `false`. Sometimes that truth value is important because you use it in a calculation, but sometimes you ignore it. Understanding how expressions evaluate to `true` or to `false` is an important part of understanding PHP.

Most scalar values are `true`. All integers and floating-point numbers (except for 0 and 0.0) are `true`. All strings are `true` except for two: a string containing nothing at all and a string containing only the character `0`. These four values are `false`. The special constant `false` also evaluates to `false`. Everything else is `true`.[1]

[1] An empty array is also `false`. This is discussed in Chapter 4.

A variable equal to one of the five `false` values, or a function that returns one of those values also evaluates to `false`. Every other expression evaluates to `true`.

Figuring out the truth value of an expression has two steps. First, figure out the actual value of the expression. Then, check whether that value is `true` or `false`. Some expressions have common sense values. The value of a mathematical expression is what you'd get by doing the math with paper and pencil. For example, `7 * 6` equals `42`. Since 42 is `true`, the expression `7 * 6` is `true`. The expression `5 - 6 + 1` equals `0`. Since `0` is `false`, the expression `5 - 6 + 1` is `false`.

The same is true with string concatenation. The value of an expression that concatenates two strings is the new, combined string. The expression '`jacob`' . '`@example.com`' equals the string `jacob@example.com`, which is `true`.

The value of an assignment operation is the value being assigned. The expression `$price = 5` evaluates to `5`, since that's what's being assigned to `$price`. Because assignment produces a result, you can chain assignment operations together to assign the same value to multiple variables:

```
$price = $quantity = 5;
```

This expression means "set `$price` equal to the result of setting `$quantity` equal to `5`." When this expression is evaluated, the integer `5` is assigned to the variable `$quantity`. The result of that assignment expression is `5`, the value being assigned. Then, that result (`5`) is assigned to the variable `$price`. Both `$price` and `$quantity` are set to `5`.

3.2 Making Decisions

With the `if( )` construct, you can have statements in your program that are only run if certain conditions are `true`. This lets your program take different actions depending on the circumstances. For example, you can check that a user has entered valid information in a web form before letting her see sensitive data.

The `if( )` construct runs a block of code if its test expression is `true`. This is demonstrated in Example 3-1.

Example 3-1. Making a decision with if( )

```
if ($logged_in) {
   print "Welcome aboard, trusted user.";
}
```

The `if( )` construct finds the truth value of the expression inside its parentheses (the *test expression*). If the expression evaluates to `true`, then the statements inside the curly braces after the `if( )` are run. If the expression isn't `true`, then the program continues with the statements after the curly braces. In this case, the test expression is just the variable

`$logged_in`. If `$logged_in` is `true` (or has a value such as `5`, `-12.6`, or `Grass Carp`, that evaluates to `true`), then `Welcome aboard, trusted user`. is printed.

You can have as many statements as you want in the code block inside the curly braces. However, you need to terminate each of them with a semicolon. This is the same rule that applies to code outside an `if( )` statement. You don't, however, need a semicolon after the closing curly brace that encloses the code block. You also don't put a semicolon after the opening curly brace. Example 3-2 shows an `if( )` clause that runs multiple statements when its test expression is `true`.

Example 3-2. Multiple statements in an if( ) code block

```
print "This is always printed.";
if ($logged_in) {
    print "Welcome aboard, trusted user.";
    print 'This is only printed if $logged_in is true.';
}
print "This is also always printed.";
```

To run different statements when the `if( )` test expression is `false`, add an `else` clause to your `if( )` statement. This is shown in Example 3-3.

Example 3-3. Using else with if( )

```
if ($logged_in) {
    print "Welcome aboard, trusted user.";
} else {
    print "Howdy, stranger.";
}
```

In Example 3-3, the first `print` statement is only executed when the `if( )` test expression (the variable `$logged_in`) is `true`. The second `print` statement, inside the `else` clause, is only run when the test expression is `false`.

The `if( )` and `else` constructs are extended further with the `elseif( )` construct. You can pair one or more `elseif( )` clauses with an `if( )` to test multiple conditions separately. Example 3-4 demonstrates `elseif( )`.

Example 3-4. Using elseif( )

```
if ($logged_in) {
    // This runs if $logged_in is true
    print "Welcome aboard, trusted user.";
} elseif ($new_messages) {
    // This runs if $logged_in is false but $new_messages is true
    print "Dear stranger, there are new messages.";
} elseif ($emergency) {
    // This runs if $logged_in and $new_messages are false
    // But $emergency is true
    print "Stranger, there are no new messages, but there is an emergency.";
}
```

If the test expression for the `if( )` statement is `true`, the PHP interpreter executes the statements inside the code block after the `if( )` and ignores the `elseif( )` clauses and their code blocks. If the test expression for the `if( )` statement is `false`, then the interpreter moves on to the first `elseif( )` statement and applies the same logic. If that test expression is `true`, then it runs the code block for that `elseif( )` statement. If it is `false`, then the interpreter moves on to the next `elseif( )`.

For a given set of `if( )` and `elseif( )` statements, at most one of the code blocks is run: the code block of the first statement whose test expression is `true`. If the test expression of the `if( )` statement is `true`, none of the `elseif( )` code blocks are run, even if their test expressions are `true`. Once one of the `if( )` or `elseif( )` test expressions is `true`, the rest are ignored. If none of the test expressions in the `if( )` and `elseif( )` statements are `true`, then none of the code blocks are run.

You can use `else` with `elseif( )` to include a code block that runs if none of the `if( )` or `elseif( )` test expressions are `true`. Example 3-5 adds an `else` to the code in Example 3-4.

Example 3-5. elseif( ) with else

```
if ($logged_in) {
    // This runs if $logged_in is true
    print "Welcome aboard, trusted user.";
} elseif ($new_messages) {
    // This runs if $logged_in is false but $new_messages is true
    print "Dear stranger, there are new messages.";
} elseif ($emergency) {
    // This runs if $logged_in and $new_messages are false
    // But $emergency is true
    print "Stranger, there are no new messages, but there is an emergency.";
} else {
    // This runs if $logged_in, $new_messages, and
    // $emergency are all false
    print "I don't know you, you have no messages, and there's no emergency.";
}
```

All of the code blocks we've used so far have been surrounded by curly braces. Strictly speaking, you don't need to put curly braces around code blocks that contain just one statement. If you leave them out, the code still executes correctly. However, reading the code can be confusing if you leave out the curly braces, so it's always a good idea to include them. The PHP interpreter doesn't care, but humans who read your programs (especially you, reviewing code a few months after you've originally written it) appreciate the clarity that the curly braces provide.

3.3 Building Complicated Decisions

The comparison and logical operators in PHP help you put together more complicated expressions on which an `if( )` construct can decide. These operators let you compare values, negate values, and chain together multiple expressions inside one `if( )` statement.

The equality operator is `= =`. It returns `true` if the two values you test with it are equal. The values can be variables or literals. Some uses of the equality operator are shown in Example 3-6.

Example 3-6. The equality operator

```
if ($new_messages == 10) {
    print "You have ten new messages.";
}

if ($new_messages == $max_messages) {
    print "You have the maximum number of messages.";
}

if ($dinner == 'Braised Scallops') {
    print "Yum! I love seafood.";
}
```

The opposite of the equality operator is `!=`. It returns `true` if the two values that you test with it are not equal. See Example 3-7.

Assignment Versus Comparison

Be careful not to use `=` when you mean `= =`. A single equals sign assigns a value and returns the value assigned. Two equals signs test for equality and return `true` if the values are equal. If you leave off the second equals sign, you usually get an `if( )` test that is always `true`, as in the following:

```
if ($new_messages = 12) {
    print "It seems you now have twelve new messages.";
}
```

Instead of testing whether `$new_messages` equals `12`, the code shown here sets `$new_messages` to `12`. This assignment returns `12`, the value being assigned. The `if( )` test expression is always `true`, no matter what the value of `$new_messages`. Additionally, the value of `$new_messages` is overwritten. One way to avoid using `=` instead of `= =` is to put the variable on the right side of the comparison and the literal on the left side, as in the following:

```
if (12 == $new_messages) {
    print "You have twelve new messages.";
}
```

The test expression above may look a little funny, but it gives you some insurance if you accidentally use `=` instead of `= =`. With one equals sign, the test expression is `12 = $new_messages`, which means "assign the value of `$new_messages` to `12`." This doesn't make any sense: you can't change the value of `12`. If the PHP interpreter sees this in your program, it reports a parse error and the program doesn't run. The parse error alerts you to the missing `=`. With the literal on the righthand side of the expression, the code is parseable by the interpreter, so it doesn't report an error.

Example 3-7. The not-equals operator

```
if ($new_messages != 10) {
    print "You don't have ten new messages.";
}

if ($dinner != 'Braised Scallops') {
    print "I guess we're out of scallops.";
}
```

With the less-than operator (<) and the greater-than operator (>), you can compare amounts. Similar to < and > are <= ("less than or equal to") and >= ("greater than or equal to"). Example 3-8 shows how to use these operators.

Example 3-8. Less-than and greater-than

```
if ($age> 17) {
    print "You are old enough to download the movie.";
}
if ($age >= 65) {
    print "You are old enough for a discount.";
}
if ($celsius_temp <= 0) {
    print "Uh-oh, your pipes may freeze.";
}
if ($kelvin_temp < 20.3) {
    print "Your hydrogen is a liquid or a solid now.";
}
```

As mentioned in Section 2.2, floating-point numbers are stored internally in such a way that they could be slightly different than their assigned value. For example, 50.0 could be stored internally as 50.00000002. To test whether two floating-point numbers are equal, check whether the two numbers differ by less than some acceptably small threshold instead of using the equality operator. For example, if you are comparing currency amounts, then an acceptable threshold would be 0.00001. Example 3-9 demonstrates how to compare two floating point numbers.

Example 3-9. Comparing floating-point numbers

```
if(abs($price_1 - $price_2) < 0.00001) {
    print '$price_1 and $price_2 are equal.';
} else {
    print '$price_1 and $price_2 are not equal.';
}
```

The `abs( )` function used in Example 3-9 returns the absolute value of its argument. With `abs( )`, the comparison works properly whether `$price_1` is larger than `$price_2` or `$price_2` is larger than `$price_1`.

The less-than and greater-than (and their "or equal to" partners) operators can be used with numbers or strings. Generally, strings are compared as if they were being looked up in a dictionary. A string that appears earlier in the dictionary is "less than" a string that appears later in the dictionary. Some examples of this are shown in Example 3-10.

Example 3-10. Comparing strings

```
if ($word < 'baa') {
    print "Your word isn't cookie.";
}
if ($word>= 'zoo') {
    print "Your word could be zoo or zymurgy, but not zone.";
}
```

String comparison can produce unexpected results, however, if the strings contain numbers or start with numbers. When the PHP interpreter sees strings like this, it converts them to numbers for the comparison. Example 3-11 shows this automatic conversion in action.

Example 3-11. Comparing numbers and strings

```
// These values are compared using dictionary order
if ("x54321"> "x5678") {
    print 'The string "x54321" is greater than the string "x5678".';
} else {
    print 'The string "x54321" is not greater than the string "x5678".';
}

// These values are compared using numeric order
if ("54321" > "5678") {
    print 'The string "54321" is greater than the string "5678".';
} else {
    print 'The string "54321" is not greater than the string "5678".';
}

// These values are compared using dictionary order
if ('6 pack' < '55 card stud') {
    print 'The string "6 pack" is less than than the string "55 card stud".';
} else {
    print 'The string "6 pack" is not less than the string "55 card stud".';
}

// These values are compared using numeric order
if ('6 pack' < 55) {
    print 'The string "6 pack" is less than the number 55.';
} else {
    print 'The string "6 pack" is not less than the number 55.';
}
```

The output of the four tests in Example 3-11 is:

```
The string "x54321" is not greater than the string "x5678".
The string "54321" is greater than the string "5678".
The string "6 pack" is not less than the string "55 card stud".
The string "6 pack" is less than the number 55.
```

In the first test, because both of the strings start with a letter, they are treated as regular strings and compared using dictionary order. Their first two characters (`x5`) are the same, but the third character of the first word (`4`) is less than the third character of the second word (`6`),[2] so the greater-than comparison returns `false`. In the second test, each string

consists entirely of numerals, so the strings are compared as numbers. The number 54,321 is larger than the number 5,678, so the greater-than comparison returns `true`. In the third test, because both strings consist of numerals and other characters, they are treated as strings and compared using dictionary order. The numeral 6 comes after 5 in the interpreter's dictionary, so the less-than test returns `false`. In the last test, the PHP interpreter converts the string `6 pack` to the number `6`, and then compares it to the number `55` using numeric order. Since `6` is less than `55`, the less-than test returns `true`.

[2] The "dictionary" that the PHP interpreter uses for comparing strings are the ASCII codes for characters. This puts numerals before letters, and orders the numerals from 0 to 9. It also puts uppercase letters before lowercase letters.

If you want to ensure that the PHP interpreter compares strings using dictionary order without any converting to numbers behind the scenes, use the built-in function `strcmp( )`. It always compares its arguments in dictionary order.

The `strcmp( )` function takes two strings as arguments. It returns a positive number if the first string is greater than the second string or a negative number if the first string is less than the first string. "Greater than" and "less than" for `strcmp( )` are defined by dictionary order. The function returns 0 if the strings are equal.

The same comparisons from Example 3-11 are shown using `strcmp( )` in Example 3-12.

Example 3-12. Comparing strings with strcmp( )

```
$x = strcmp("x54321","x5678");
if ($x > 0) {
    print 'The string "x54321" is greater than the string "x5678".';
} elseif ($x < 0) {
    print 'The string "x54321" is less than the string "x5678".';
}

$x = strcmp("54321","5678");
if ($x > 0) {
    print 'The string "54321" is greater than the string "5678".';
} elseif ($x < 0) {
    print 'The string "54321" is less than the string "5678".';
}

$x = strcmp('6 pack','55 card stud');
if ($x > 0) {
    print 'The string "6 pack" is greater than than the string "55 card stud".';
} elseif ($x < 0) {
    print 'The string "6 pack" is less than the string "55 card stud".';
}

$x = strcmp('6 pack',55);
if ($x > 0) {
    print 'The string "6 pack" is greater than the number 55.';
} elseif ($x < 0) {
    print 'The string "6 pack" is less than the number 55.';
}
```

The output from Example 3-12 is as follows:

```
The string "x54321" is less than the string "x5678".
The string "54321" is less than the string "5678".
```

```
The string "6 pack" is greater than than the string "55 card stud".
The string "6 pack" is greater than the number 55.
```

Using `strcmp( )` and dictionary order produces different results than Example 3-11 for the second and fourth comparisons. In the second comparison, `strcmp( )` computes that the string `54321` is less than `5678` because the second characters of the strings differ and `4` comes before `6`. It doesn't matter to `strcmp( )` that `5678` is shorter than `54321` or that it is numerically smaller. In dictionary order, `54321` comes before `5678`. The fourth comparison turns out differently because `strcmp( )` doesn't convert `6 pack` to a number. Instead, it compares `6 pack` and `55` as strings and computes that `6 pack` is bigger because its first character, `6`, comes later in the dictionary than the first character of `55`.

To negate a truth value, use `!`. Putting `!` before an expression is like testing to see whether the expression equals `false`. The two `if( )` statements in Example 3-13 are equivalent.

Example 3-13. Using the negation operator

```
// The entire test expression ($finished == false)
// is true if $finished is false
if ($finished == false) {
    print 'Not done yet!';
}

// The entire test expression (! $finished)
// is true if $finished is false
if (! $finished) {
    print 'Not done yet!';
}
```

You can use the negation operator with any value. If the value is `true`, then the combination of it with the negation operator is `false`. If the value is `false`, then the combination of it with the negation operator is `true`. Example 3-14 shows the negation operator at work with a call to `strcasecmp( )`.

Example 3-14. Negation operator

```
if (! strcasecmp($first_name,$last_name)) {
    print '$first_name and $last_name are equal.';
}
```

In Example 3-14, the statement in the `if( )` code block is executed only when the entire test expression is `true`. When the two strings provided to `strcasecmp( )` are equal (ignoring capitalization), `strcasecmp( )` returns 0, which is `false`. The test expression is the negation operator applied to this `false` value. The negation of `false` is `true`. So, the entire test expression is `true` when two equal strings are given to `strcasecmp( )`.

With logical operators, you can combine multiple expressions inside one `if( )` statement. The logical AND operator, `&&`, tests whether one expression and another are both `true`. The logical OR operator, `||`, tests whether either one expression or another is `true`. These logical operators are used in Example 3-15.

Example 3-15. Logical operators

```
if (($age >= 13) && ($age < 65)) {
   print "You are too old for a kid's discount and too young for the senior's discount.";
}

if (($meal == 'breakfast') || ($dessert == 'souffle')) {
   print "Time to eat some eggs.";
}
```

The first test expression in Example 3-15 is `true` when both of its subexpressions are `true` — when `$age` is at least 13 but not more than 65. The second test expression is `true` when either of its subexpressions are `true` — when `$meal` is `breakfast` or `$dessert` is `souffle`.

The admonition about operator precedence and parentheses from Chapter 2 holds true for logical operators in test expressions, too. To avoid ambiguity, surround with parentheses each subexpression inside a larger test expression.

3.4 Repeating Yourself

When a computer program does something repeatedly, it's called *looping*. This happens a lot — for example, when you want to retrieve a set of rows from a database, print rows of an HTML table, or print elements in an HTML `<select>` menu. The two looping constructs discussed in this section are `while( )` and `for( )`. Their specifics differ but they each require you to specify the two essential attributes of any loop: what code to execute repeatedly and when to stop. The code to execute is a code block just like what goes inside the curly braces after an `if( )` construct. The condition for stopping the loop is a logical expression just like an `if( )` construct's test expression.

The `while( )` construct is like a repeating `if( )`. You provide an expression to `while( )`, just like to `if( )`. If the expression is `true`, then a code block is executed. Unlike `if( )`, however, `while( )` checks the expression again after executing the code block. If it's still `true`, then the code block is executed again (and again, and again, as long as the expression is `true`.) Once the expression is `false`, program execution continues with the lines after the code block. As you have probably guessed, your code block should do something that changes the outcome of the test expression so that the loop doesn't go on forever.

Example 3-16 uses `while( )` to print out an HTML form `<select>` menu with 10 choices.

Example 3-16. Printing a <select> menu with while( )

```
$i = 1;
print '<select name="people">';
while ($i <= 10) {
    print "<option>$i</option>\n";
    $i++;
}
print '</select>';
```

Example 3-16 prints:

```
<select name="people"><option>1</option>
<option>2</option>
```

```
<option>3</option>
<option>4</option>
<option>5</option>
<option>6</option>
<option>7</option>
<option>8</option>
<option>9</option>
<option>10</option>
</select>
```

Before the `while( )` loop runs, the code sets `$i` to `1` and prints the opening `<select>` tag. The test expression compares `$i` to `10`. As long as `$i` is less than or equal to 10, the two statements in the code block are executed. The first prints out an `<option>` tag for the `<select>` menu, and the second increments `$i`. If you didn't increment `$i` inside the `while( )` loop, Example 3-16 would print out `<option>1</option>` forever.

After the code block prints `<option>10</option>`, the `$i++` line makes `$i` equal to 11. Then the test expression (`$i <= 10`) is evaluated. Since it's not `true` (11 is not less than or equal to 10), the program continues past the `while( )` loop's code block and prints out the closing `</select>` tag.

The `for( )` construct also provides a way for you to execute the same statements multiple times. Example 3-17 uses `for( )` to print out the same HTML form `<select>` menu as Example 3-16.

Example 3-17. Printing a <select> menu with for( )

```
print '<select name="people">';
for ($i = 1; $i <= 10; $i++) {
    print "<option>$i</option>";
}
print '</select>';
```

Using `for( )` is a little more complicated than using `while( )`. Instead of one test expression in parentheses, there are three expressions, separated with semicolons: the initialization expression, the test expression, and the iteration expression. Once you get the hang of it, however, `for( )` is a more concise way to have a loop with easy-to-express initialization and iteration conditions.

The first expression in Example 3-17 (`$i = 1`) is the *initialization expression*. It is evaluated once when the loop starts. This is where you put variable initializations or other setup code. The second expression in Example 3-17 (`$i <= 10`)) is the test expression. It is evaluated once each time through the loop, before the statements in the loop body. If it's `true`, then the loop body is executed (`print "<option>$i</option>";` in Example 3-17). The third expression in Example 3-17 (`$i++`) is the *iteration expression*. It is run after each time the loop body is executed. In Example 3-17, the sequence of statements goes like this:

1. Initialization expression: `$i = 1;`
2. Test expression: `$i <= 10` (`true`, `$i` is `1`)
3. Code block: `print "<option>$i</option>";`
4. Iteration expression: `$i++;`
5. Test expression: `$i <= 10` (`true`, `$i` is `2`)

6. Code block: `print "<option>$i</option>";`
7. Iteration expression: `$i++;`
8. (Loop continues with incrementing values of `$i`)
9. Test expression: `$i <= 10` (`true`, `$i` is `9`)
10. Code block: `print "<option>$i</option>";`
11. Iteration expression: `$i++;`
12. Test expression: `$i <= 10` (`true`, `$i` is `10`)
13. Code block: `print "<option>$i</option>";`
14. Iteration expression: `$i++;`
15. Test expression: `$i <= 10` (`false`, `$i` is `11`)

You can combine multiple expressions in the initialization expression and the iteration expression of a `for( )` loop by separating each of the individual expressions with a comma. This is usually done when you want to change more than one variable as the loop progresses. Example 3-18 applies this to the variables `$min` and `$max`.

Example 3-18. Multiple expressions in for( )

```
print '<select name="doughnuts">';
for ($min = 1, $max = 10; $min < 50; $min += 10, $max += 10) {
    print "<option>$min - $max</option>\n";
}
print '</select>';
```

Each time through the loop, `$min` and `$max` are each incremented by 10. Example 3-18 prints:

```
<select name="doughnuts"><option>1 - 10</option>
<option>11 - 20</option>
<option>21 - 30</option>
<option>31 - 40</option>
<option>41 - 50</option>
</select>
```

3.5 Chapter Summary

Chapter 3 covers:

- Evaluating an expression's truth value: `true` or `false`.
- Making a decision with `if( )`.
- Extending `if( )` with `else`.
- Extending `if( )` with `elseif( )`.
- Putting multiple statements inside an `if( )`, `elseif( )`, or `else` code block.
- Using the equality (`= =`) and not-equals (`!=`) operators in test expressions.
- Distinguishing between assignment (`=`) and equality comparison (`= =`).
- Using the less-than (`<`), greater-than (`>`), less-than-or-equal-to (`<=`), and greater-than-or-equal-to (`>=`) operators in test expressions.
- Comparing two floating-point numbers with `abs( )`.
- Comparing two strings with operators.

- Comparing two strings with `strcmp( )` or `strcasecmp( )`.
- Using the negation operator (`!`) in test expressions.
- Using the logical operators (`&&` and `||`) to build more complicated test expressions.
- Repeating a code block with `while( )`.
- Repeating a code block with `for( )`.

3.6 Exercises

1. Without using a PHP program to evaluate them, determine whether each of these expressions is `true` or `false`:
    a. `100.00 - 100`
    b. `"zero"`
    c. `"false"`
    d. `0 + "true"`
    e. `0.000`
    f. `"0.0"`
    g. `strcmp("false","False")`
2. Without running it through the PHP interpreter, figure out what this program prints.

```
  $age = 12;
  $shoe_size = 13;
  if ($age > $shoe_size) {
      print "Message 1.";
  } elseif (($shoe_size++) && ($age > 20)) {
      print "Message 2.";
  } else {
      print "Message 3.";
  }
print "Age: $age. Shoe Size: $shoe_size";
```

12. Use `while( )` to print out a table of Fahrenheit and Celsius temperature equivalents from -50 degrees F to 50 degrees F in 5-degree increments. On the Fahrenheit temperature scale, water freezes at 32 degrees and boils at 212 degrees. On the Celsius scale, water freezes at 0 degrees and boils at 100 degrees. So, to convert from Fahrenheit to Celsius, you subtract 32 from the temperature, multiply by 5, and divide by 9. To convert from Celsius to Fahrenheit, you multiply by 9, divide by 5, and then add 32.
13. Modify your answer to Exercise 3 to use `for( )` instead of `while( )`.