Programming via PHP Simple programs

3.1. A first program

Let us begin with a very simple PHP program. Create a file (named `random.php` perhaps) that says the following.

```
<html>
<head><title>Random number</title></head>
<body>
<p>I have chosen <?php
    echo rand(1, 100);
?>.</p>
</body>
</html>
```

You can install it on the Web server; if the Web server supports PHP, then when you load it into your Web browser you'll see something like the following.

```
I have chosen 47.
```

Let's try to make sense of this simple program. The first thing you'll notice is that our PHP program is basically HTML. In fact, when the Web server sees a PHP file, it sends the page to the Web browser *verbatim* until it finds the character sequence `<?php`. Thus, when the browser requests the file, the server starts to read through the file and simply sends each character along to it; but when it sees the five-character sequence `<?php`, it thinks, Uh-oh. Here comes some PHP that tells me that I have to do some work before sending any more information to the browser.

In this program, the server would see the statement `echo rand(1, 100);`, which as we'll see later means that the Web server should send a random number between 1 and 100 to the browser. Then it sees `?>`, whereupon the server thinks, Whew! I'm past all that work. Now I can go back to sending characters without having to think about it. So it goes along.

The overall result of the above process is that when a Web browser accesses the Web page, the server actually sends the following HTML to the browser.

```
<html>
<head><title>Random Number</title></head>
```

```
<body>
<p>I have chosen 47.</p>
</body>
</html>
```

Notice that the Web page that the browser receives straight HTML, just as if the Web page were a simple HTML file. Because the browser never sees any PHP, there is no reason for us as PHP programmers to worry about browser support: As long as the browser supports HTML, it will be able to read our PHP pages. (This is not true of pages using client-side programming technologies, such as JavaScript or Flash, where Web browsers will need to be able to read those programs in addition to regular HTML.)

3.2. The `echo` statement

PHP programs consist of a sequence of **statements**, which typically end with a semicolon. In processing PHP code, the server processes the first statement; then it goes to the next statement; then the next; then the next; until it reaches the end of the PHP. (You can see why a semicolon is appropriate for ending each individual statement.)

The PHP portion of the above Web page contains just a single statement: **`echo`** `rand(1, 100);`. This is an example of an **`echo`** statement. When the Web server reaches an **`echo`** statement, it knows that it will send some information to the Web browser. Whatever comes between the word **`echo`** and the semicolon is what it should send; in this case, it sees that it should whatever number `rand` generates.

The bit between **`echo`** and the semicolon are a bit odd: What is going on here is that we are using something called a **function**. You probably remember seeing functions in algebra class; for example, you may have defined $f(x)$ to be $\frac{1}{2} x^2 + 2x + 1$, and then you could compute $f(2)$ (which would be 7 in this case). We would call x the **parameter** for the function f.

It happens that `rand` is one of the functions in the vast library defined by PHP. Whenever we want to use the function, we need to include a set of parentheses after it (just as we had to do with f in our algebra class); in `rand`'s case, the function wants two parameters, which specify the range from which to select a random number.

3.3. Variables

Sometimes in writing a program, we want the computer to temporarily remember a value so that it can be used later on. Suppose, for example, that we want the computer to select a random number and display its square root. We would want to use the number in two different **echo** statements: In one, we want to display the chosen number; and then later, we want to display that number's square root. We need a way to remember the selected value from one statement to the next.

To do this, we use a programming concept called a **variable** — it is simply a way of assigning a name to a value. In PHP, variable names must always begin with a dollar sign ('`$`').

To create a variable, we use an **assignment statement** — the second type of statement that we've seen (the first was the **echo** statement). Here is a simple example.

```
$pi = 3.1415;
```

An assignment statement begins with the variable name, followed by an equals sign, followed by the value to give it; and it ends with a semicolon. In this case, the variable I am creating is named `$pi`, and I am assigning the name to refer to the value 3.1415.

Now we can look at the PHP program to choose a random number and display its square root.

```
<html>
<head><title>Random</title></head>
<body>
<p>I have randomly selected the number <?php
    $choice = rand(1, 100);
    echo $choice;
?>. Its square root is <?php
    echo sqrt($choice);
?>.</p>
</body>
</html>
```

When the Web server reaches the PHP code, it will see that it is supposed to accomplish an assignment statement: It says to assign the variable `$choice` to refer to whatever value the `rand` function happens to generate.

Having completed that assignment statement, it goes onto the next statement. It sees that the second statement is an **echo** statement, which says to send `$choice` to the Web browser.

When it then reaches the two-character sequence `?>`, the server realizes that it has reached the end of the PHP code, and it begins sending letters to the Web browser as part of the HTML. But it soon reaches the `<?php` sequence again, which means that it should go back to interpreting PHP.

In this case, it finds another **echo** statement to execute; in this case, it is supposed to call the `sqrt` function, passing the value of `$choice` as a parameter. As an **echo**statement, it will send the `sqrt` function's result to the browser.

The upshot of all this is that the user will see the following Web page, if it happens that the Web server ends up choosing the number 64.

I have randomly selected the number 64. Its square root is 8.

If the user then reloads the page, the Web server would probably choose a different number between 1 and 100 and display it and its square root.

It's instructive to consider the result of changing our program in a couple of ways. Let me ask them as questions first, to give you a chance to think about each before seeing the answers below.

1. Suppose we were to eliminate the statement **echo** `$choice;` from our PHP program. What would happen?
2. Suppose we try to eliminate the `$choice` variable by instead writing:

   ```
   <p>I have randomly selected the number <?php
       echo rand(1, 100);
   ?>. Its square root is <?php
       echo sqrt(rand(1, 100));
   ?>.</p>
   ```

   What would happen then?

The answer to our first question is that if we omitted the **echo** `$choice;` statement, then the Web server would never know that the value of `$choice` should be sent to the Web server. A user who loaded the page would see something like the following.

I have randomly selected the number . Its square root is 8.

Notice: *When a Web server executes an assignment statement, it doesn't send anything to the Web browser. It simply remembers the value for future use.* This may not be the intuitive behavior to you, but it turns out to more useful for the assignment statement to work this way.

For our second question, the Web server would end up asking the `rand` function for a value two different times. Now it happens that the `rand` function responds with a different value each time it is asked (unless by some weird coincidence it chooses the same value both times). So the user may see something like the following (if `rand`happens to respond with 42 the first time and 4 the second time).

```
I have randomly selected the number 42. Its square root is 2.
```

3.4. White space and capitalization

One thing you'll notice in the above examples is that I've been fairly particular about indenting each piece of PHP code and placing each statement on its own line. Actually, lines breaks and spaces are irrelevant in PHP (just as the are irrelevant in most HTML). I could instead have written our random square-root program as the following, and it would work just as well.

```
<html>
<head><title>Random</title></head>
<body>
<p>I have randomly selected the number
<?php $choice=rand(1,100); echo $choice;?>.
Its square root is <?php echo sqrt($choice);?>.</p>
</body>
</html>
```

However, you should adopt the habit of placing each statement on its separate line, because that makes your program easier to read and to edit. In fact, this convention of breaking lines with each statement is so strong among programmers that they use the term **line** as a synonym for *statement*. Later, we'll see that sometimes statements can grow very long, so that it's nice to break it across multiple lines; even then, though, a programmer might refer to the set of several lines constituting a single statement as a line.

(Indenting your PHP is less important. I've been doing it just to emphasize its separation from the regular HTML. When I write my own PHP, though, I often don't indent the PHP.)

Another important point in PHP concerns capitalization: The capitalization of letters is sometimes significant in PHP. (Sometimes it's not significant (as with function names), but it's easier not to worry about the exact rules.) Thus, the names `$x` and `$X` refer to two completely different variables. Generally speaking, the easiest thing to name your variables using all lower-case letters so that you don't have to think about capitalization.

Source: http://www.toves.org/books/php/ch03-first/index.html