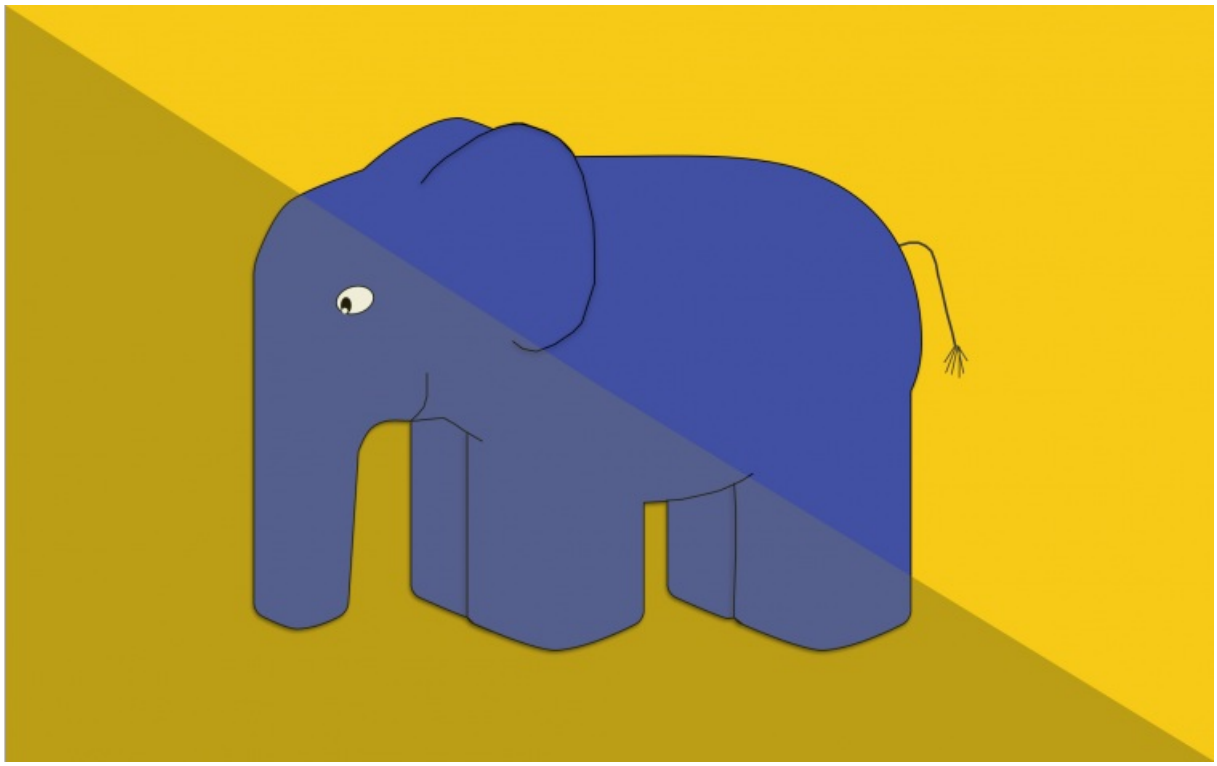
Course 1: How to win friends & develop in PHP



With <3 from SymfonyCasts

Chapter 1: Let's Write some PHP!

LET'S WRITE SOME PHP!¶

Welcome! We're glad you're here with us to learn how to become an Epic PHP developer. PHP is a programming language that runs a large percentage of the web including sites as big as Facebook. But since PHP has been around for awhile, there is a lot of bad, outdated and boring information about PHP on the web.

But not here! In this course, we'll learn PHP from scratch by building a real website. This means you'll learn the practices used by real *employed* developers to build really cool things, and not just a bunch of theory. We'll teach you something in each chapter and then you'll test and practice your new skills by coding right in your browser. Learn and then practice, that's the key! Before long, you'll be creating more and more complex things and be the coolest guy or gal that any of your friends know - probably :)

The Project¶

We're going to build a site that we're calling `AirPupNMeow.com`. Imagine a site like Airbnb.com, except where people rent cute pets instead of apartments. If you're looking for companionship without all that responsibility of walking your dog every morning and bringing a bag to pick up his... uh gifts, then this site would be for you! Ok, the idea might be kinda silly, but that hasn't stopped startups in the past! So let's go!

What you see here is just an HTML page that I've loaded in my browser

> http://localhost:8000/index.php

This is a template based on Twitter Bootstrap and it's just a bunch of hardcoded text and links that don't go anywhere yet. But it's already a cute start to our rent-a-pet site.

For now, don't worry about the `localhost` part I have in the URL or that this file ends in `.php`. Just know that when I load this page, the `index.php` file is being opened and all the HTML is rendered by my browser.

I'm going to use my favorite editor called `PhpStorm` to open this file and prove that it's only simple HTML. Later in this course, we'll get your computer setup to run and modify files just like this.

Right now, this page is totally static. Each time I refresh the page, I get back the same exact HTML. On real websites, things are dynamic: news stories update when I refresh and personalized information is pulled from a database. That's the kind of stuff that PHP does.

Writing your first PHP¶

Let's make this page more interesting!

Before you write PHP code, you'll always start with the same opening tag: `<?php`. This is what tells PHP that we're not writing HTML anymore - we actually want to write some PHP code. Let's print out a cool message by using the `echo` statement and surrounding our message with single quotes. Finish off the line with a semicolon and then write the PHP closing tag: `?>`. These last two characters get us out of PHP mode and back into HTML. The `<?php` and `?>` tags are exact opposites and always come in a pair. One gets us into PHP mode and the other exits PHP mode:

```
<!-- index.php -->
<!-- ... -->

<div class="jumbotron">
  <div class="container">
    <h1><?php echo 'Look mom, PHP!'; ?></h1>

    <!-- ... -->
  </div>
</div>
```

Before we talk about what we did, let's celebrate, because when I refresh the page, it works! PHP is printing our message in the middle of the page.

The key is the `echo` statement, whose job is to print things out. The message itself is called a "string" and strings are always surrounded by single quotes when you write them.

Tip

You can actually use single quotes (`'Foo'`) or double quotes (`"Foo"`). They're basically the same. But using single quotes is much more hipster!

Creating and Using Variables¶

Since printing a static string is boring, let's create a variable! Whenever we want to write PHP code, remember to open up PHP with `<?php` and close it with `?>`:

```
<!-- index.php -->
<!-- ... -->

<div class="jumbotron">
  <div class="container">
    <?php

    ?>

    <h1><?php echo 'Look mom, PHP!'; ?></h1>

    <!-- ... -->
  </div>
</div>
```

The open and close PHP tags can totally be on separate lines. If we refresh now, there's no change. Unless we print something from within PHP, nothing is shown on the page. Even if we add blank lines, they don't appear inside the HTML source code.

To create a variable, start with a dollar sign (`$`), write a clever name, then finish it up with an equal sign (`=`) and the value we want to give, or assign, to the variable. Remember to add a semi-colon at the end of the line: almost all lines in PHP end in a semi-colon. Did you hear me? Because, forgetting this is one of the most common errors you'll make:

```
<?php
  $cleverWelcomeMessage = 'All the love, none of the crap!';
?>
```

If we refresh, nothing changes yet. that makes sense, because we haven't printing anything from within PHP! Using the variable is easy, replace our echo'd string with a `$` and the variable name. and just like that, we're creating and using variables and one step closer to your new best friend:

```
<!-- index.php -->
<!-- ... -->

<div class="jumbotron">
  <div class="container">
    <?php
      $cleverWelcomeMessage = 'All the love, none of the crap!';
    ?>

    <h1><?php echo $cleverWelcomeMessage; ?></h1>

    <!-- ... -->
  </div>
</div>
```

Variables as Strings or Numbers¶

Of course, variables can also be set to numbers which looks the same but without the quotes:

```
<?php
  $cleverWelcomeMessage = 'All the love, none of the crap!';
  $pupCount = 5000
?>
```

Notice that I have 2 PHP lines, or statements, inside one set of opening and closing PHP tags. That's totally legal: once you open PHP, you can write as much as you want. Use your new variable to print another message:

```
<!-- index.php -->
<!-- ... -->

<div class="jumbotron">
  <div class="container">
    <?php
      $cleverWelcomeMessage = 'All the love, none of the crap!';
      $pupCount = 5000;
    ?>

    <h1><?php echo $cleverWelcomeMessage; ?></h1>

    <p>With over <?php echo $pupCount ?> pet friends!</p>
    <!-- ... -->
  </div>
</div>
```

When we fresh, it's a success!

Making PHP Angry with Syntax Errors!¶

Now, let's make a small error to see what happens. I'll just remove the semicolon from the end of the `$cleverWelcomeMessage` line:

```
<!-- index.php -->
<!-- ... -->

<div class="jumbotron">
  <div class="container">
    <?php
      $cleverWelcomeMessage = 'All the love, none of the crap!'
      $pupCount = 5000;
    ?>

    <h1><?php echo $cleverWelcomeMessage; ?></h1>

    <p>With over <?php echo $pupCount ?> pet friends!</p>
    <!-- ... -->
  </div>
</div>
```

> PHP Parse error: syntax error, unexpected '$pupCount' (T_VARIABLE) in /path/to/site/index.php on line 70

You'll see a lot of error messages and the trick is to get good at knowing what they mean. Be sure to look at the line number and check that line *and* the lines *above* it. In this case, the error is being reported in the line with `$pupCount`. But there's nothing wrong with this line - the missing semicolon is actually the line *above* this. That's really common with PHP errors, so look for it!

Ok, now it's your turn! Test out your skills with the activities!

Chapter 2: Functions!

FUNCTIONS!¶

We already know what to write when we want to use some PHP code, how to set a variable, and how to print things. Like most languages, PHP also has functions, like `rand`, which gives us a random number:

```
<!-- index.php -->
<!-- ... -->

<?php
  $cleverWelcomeMessage = 'All the love, none of the crap!';
  $pupCount = rand();
?>
```

A function always starts with its name followed by opening and closing parentheses. Those are the key and tell PHP that `rand` is a function. Think of a function like a machine: we execute it with the code you see here and it returns a value. Not all functions work exactly like this, but most do. So when you see a function, think "this does some work for me and then returns a value." It might return a number, a string like `'Hello World'`, or something more complicated.

The `rand` function returns a random number. We assign that number to the `$pupCount` variable and then print it just like before with the `echo` statement.

When we refresh, we see our number is now dynamic: each time we refresh, the `rand` function gives us another number!

Tip

Generating random numbers is actually kind of tough for computers. Another function, :phpfunction`mt_rand`, generates "better" random numbers.

PHP comes with a lot of built-in functions like `rand` that you can use immediately. On php.net, you can lookup your function and learn all about it.

Tip

At the time of this recording, php.net was getting a facelift! If you see an older, uglier site, you may see a link at the top of the page to preview the newer site.

Functions with Arguments¶

Sometimes we can control the behavior of a function by passing it an argument:

```
<!-- index.php -->
<!-- ... -->

<?php
  $cleverWelcomeMessage = 'All the love, none of the crap!';
  $pupCount = rand(50);
?>
```

An argument appears between the parentheses of the function and tells the `rand` function to give us a number that's 50 or larger. For `rand`, this argument is optional: the function will work without it and has a default value of `0`. I know this by reading its documentation.

Tip

PHP often uses the word "parameter" in place of argument in its documentation and error messages. These two words mean the same thing.

In fact, we can see that `rand` has *2* arguments: the minimum number *and* a maximum. To pass a second argument, just add a comma after the first:

```
<!-- index.php -->
<!-- ... -->

<?php
  $cleverWelcomeMessage = 'All the love, none of the crap!';
  $pupCount = rand(50, 100);
?>
```

When we refresh, our pup number is random, but between 50 and 100. Functions are machines that do work and return a value. Arguments are input that let us control the function. We pass arguments to the function as a comma-separated list inside its parentheses.

Tip

Functions don't always return a value. Some functions just *do* something but return nothing. An example is `var_dump`, which prints to the screen similar to `echo`. We'll see this in a moment.

Capitalizing the first Letter of each Word¶

Every function has a different number of total arguments that mean different things. Let's look up a cool function called `ucwords`. This function has only one argument, but it's required:

```
<!-- index.php -->
<!-- ... -->

<?php
  $cleverWelcomeMessage = ucwords('All the love, none of the crap!');
  $pupCount = rand(50, 100);
?>
```

When we refresh the browser, every word in the string is upper-cased!

```
All The Love, None Of The Crap!
```

Since the one argument is required, if we leave it off, PHP will give us a "friendly" reminder:

```
<!-- index.php -->
<!-- ... -->

<?php
  $cleverWelcomeMessage = ucwords();
  $pupCount = rand(50, 100);
?>
```

```
PHP Warning: ucwords() expects exactly 1 parameter, 0 given in /path/to/project/index.php on line 69
```

The point is that PHP has *a lot* of functions, and each has different arguments that mean different things. Some arguments are required, like the first and only argument of `ucwords` and some are optional, like both arguments to `rand`.

When you need to do something like generate a random number, the best thing to do is google your question, find the function you need, then research it on php.net. Every page has comments below it and a spot where you can learn about similar functions.

Lowercasing all letters / Using Functions in Different Places¶

Let's look at one of the related functions `strtolower`. Like the name suggests, when we give this function its one required argument, it will make every character lowercase and return it. Let's replace `ucwords` with this. But instead of using it to set

the `$cleverWelcomeMessage` variable to a lowercase string, we can use it to lowercase the string message just before `echo` prints it:

```
<!-- index.php -->
<!-- ... -->

<div class="jumbotron">
  <div class="container">
    <?php
      $cleverWelcomeMessage = 'All the love, none of the crap!';
      $pupCount = rand(50, 100);
    ?>

    <h1><?php echo strtolower($cleverWelcomeMessage); ?></h1>
    <!-- ... -->
  </div>
</div>
```

Just like your new pup, a function can really go anywhere. And variables can be used as arguments. Remember, `$cleverWelcomeMessage` represents our string message, so this is the same as passing the string directly (e.g. `strtolower('All the love, none of the crap!')`).

Let's get fancy and use another function - `strrev` - to print the string in reverse:

```
<h1><?php echo strrev(strtolower($cleverWelcomeMessage)); ?></h1>
```

When we fresh, our string is all lowercase AND reversed.

```
<h1>!parc eht fo enon ,evol eht lla</h1>
```

You can use functions inside of functions like this as much as you want. The trick is to keep track of your parenthesis and always remember to have a closing parenthesis for every opening one.

But what order do things take place? Is the string lowercased and then reversed or reversed first and then lowercased? If we replace `strrev` with `strtoupper`, the opposite of `strtolower`, then it becomes obvious:

```
<h1><?php echo strtoupper(strtolower($cleverWelcomeMessage)); ?></h1>
```

When we refresh, the string displays completely in upper case:

```
<h1>ALL THE LOVE, NONE OF THE CRAP!</h1>
```

This proves that the string is lowercased first and *then* uppercased. Functions work from the inside out. Initially `cleverWelcomeMessage` is passed as the first argument to `strtolower` and a lowercase string is returned. This lowercase string is then passed as the first argument to `strtoupper`, which returns an upper case string. Which is finally printed with `echo`. Phew!

This is all really cool, but if you do feel overwhelmed, you could always write this using multiple lines:

```
<!-- index.php -->
<!-- ... -->

<div class="jumbotron">
  <div class="container">
    <?php
      $cleverWelcomeMessage = 'All the love, none of the crap!';
      $lowerMessage = strtolower($cleverWelcomeMessage);
      $upperMessage = strtoupper($lowerMessage);
      $pupCount = rand(50, 100);
    ?>

    <h1><?php echo $upperMessage; ?></h1>
    <!-- ... -->
  </div>
</div>
```

The most important thing to remember is that PHP has a lot of functions, which are always written with a set of parenthesis after their name. Some have one or more arguments that allow you to control the function and the documentation explains these. Functions typically do some work and return a value, which you can assign to variables or print using echo. Got it? Ok, onto practicing with the activities!

Chapter 3: Arrays and Loops

ARRAYS AND LOOPS¶

We've seen how we can create variables and set each to a string or a number. We've also used functions like `rand` and `strtoupper` to return numbers and strings. Let's talk about a third type of variable in PHP: an array. An array represents a group of things, like 5 random numbers or 3 strings. They're a really important type of data in PHP, and in a few minutes, you'll know all about them.

To understand arrays, first imagine that there are 3 pets in the system. For starters, I'll just set their names to 3 variables and print each out manually:

```
<!-- index.php -->
<!-- ... -->

<?php
  $pet1 = 'Chew Barka';
  $pet2 = 'Spark Pug';
  $pet3 = 'Pico de Gato';
?>

<div class="container">
  <div class="row">
    <div class="col-lg-4">
      <h2><?php echo $pet1; ?></h2>
    </div>
    <div class="col-lg-4">
      <h2><?php echo $pet2; ?></h2>
    </div>
    <div class="col-lg-4">
      <h2><?php echo $pet3; ?></h2>
    </div>
  </div>
</div>
```

When we refresh, we see the 3 park friendly pets in our styled list. Now obviously, we're repeating code unnecessarily and writing all the `div` and `h2` tags is too much work. Arrays to the rescue!

Creating an Array¶

To create an array, just say - amazingly - `array` and set it to a variable:

```
<?php
  $pet1 = 'Chew Barka';
  $pet2 = 'Spark Pug';
  $pet3 = 'Pico de Gato';

  $pets = array();
?>
```

Tip

In PHP 5.4 and higher, you can also say `$pets = [];`. The two ways mean exactly the same thing.

This looks just like a function, and basically it is. But instead of returning a string or number, it returns an array, which is like a box that you can put strings, numbers or other things into. Right now the box is empty.

To put things in the box, just pass them as arguments when creating the array:

```
<!-- index.php -->
<!-- ... -->

<?php
  $pet1 = 'Chew Barka';
  $pet2 = 'Spark Pug';
  $pet3 = 'Pico de Gato';

  $pets = array($pet1, $pet2, $pet3);
?>
```

Note

`array()` takes an unlimited number of arguments. So put as many things into your array as you want!

There are now 3 strings inside our array waiting to see what we'll do with them. Like any box full of stuff, we can either pull out one specific item or pull out every item one at a time. In our case, we want to loop through each pet and print its name inside our HTML markup.

Looping over an Array¶

In PHP, we loop over arrays using the all-important `foreach` statement. Let's remove the repeated code we have now and create just 1 loop that will print all 3:

```
<!-- index.php -->
<!-- ... -->

<?php
  $pet1 = 'Chew Barka';
  $pet2 = 'Spark Pug';
  $pet3 = 'Pico de Gato';

  $pets = array($pet1, $pet2, $pet3);
?>

<div class="container">
  <div class="row">
    <?php
      foreach ($pets as $pet) {
        echo '<div class="col-lg-4">';
        echo '<h2>';
        echo $pet;
        echo '</h2>';
        echo '</div>';
      }
    ?>
  </div>

  <!-- ... -->
</div>
```

Refresh and success! Congratulations on creating and looping over your first array. This is one of the most common and important skills in PHP.

One unfortunate side effect is that we're now printing out the static HTML tags like the `div` and the `h2` via PHP. This is totally fine, but it looks a bit uglier. There is a way to make this all look a lot prettier, which we'll talk about in the next episode of this series.

`foreach` isn't a function, it's what's called a "language construct". That basically means that it looks and works like a function, but has its own, special syntax. There aren't many of these language constructs and I'll point them out along the way.

To loop, we say `$pets as $pet`. The first variable is the array we're looping over and the second is a new variable name, which PHP sets to the value of each item in the array as we loop. Since our array has 3 strings in it, `foreach` executes the

lines between `{` and `}` 3 times and `$pet` is set to a different string each time.

If I change `$pet` to something else, that's fine, as long as I change it inside the curly braces as well:

```
foreach ($pets as $cutePet) {
    echo '<div class="col-lg-4">';
    echo '<h2>';
    echo $cutePet;
    echo '</h2>';
    echo '</div>';
}
```

Also notice that our new code contains the first 2 lines of PHP that *don't* end in a semicolon. This is pretty common: either you're writing a normal line that ends in a semi-colon or you're using a language construct that has an opening `{` and a closing `}`. We'll see another example of that later with the `if` statement.

Accessing Specific Items in an Array¶

In addition to looping over each item in an array, you can also just access one specific item. First, let's see how an array looks under the surface by using a handy debugging function called `var_dump`. `var_dump`, like `echo`, prints things to the screen. But `var_dump` is better for debugging because it prints things out in a really descriptive way. If you have a variable and want to know everything about it, `var_dump` is your new best friend:

```
<!-- index.php -->
<!-- ... -->

<?php
    $pet1 = 'Chew Barka';
    $pet2 = 'Spark Pug';
    $pet3 = 'Pico de Gato';

    $pets = array($pet1, $pet2, $pet3);
    var_dump($pets);
?>
```

When we refresh, we see the word "array" that tells us what type of value our variable is. Afterwards, we see our 3 strings next to the number 0, 1 and 2:

```
array(3) {
  [0] =>
  string(10) "Chew Barka"
  [1] =>
  string(9) "Spark Pug"
  [2] =>
  string(12) "Pico de Gato"
}
```

Array Keys/Indexes¶

As we can see, an array does more than just hold things, it also gives each a unique identifier. Imagine you're going to see a fancy orchestra performance. When you walk in, there's a coat room. You give your coat to the attendee who attaches a unique number to it and then gives you a copy of that number. The coatroom is an array of different coats, but each has a unique number. When the night is over, you tell the attendee your number and he finds and returns just your coat.

A PHP array is just as simple and as you can see, the first item is assigned the number 0, the second is assigned 1, the third item 2, and so on if we had more pets. This number is called the array key or index. Later we'll see how we can even control these keys instead of letting them be auto-assigned like it is now.

To access a single item, just look it up by its key using a square bracket syntax:

```
<!-- index.php -->
<!-- ... -->

<?php
  $pet1 = 'Chew Barka';
  $pet2 = 'Spark Pug';
  $pet3 = 'Pico de Gato';

  $pets = array($pet1, $pet2, $pet3);

  echo $pets[0];
  echo $pets[2];
?>
```

When we fresh, we see the first and third pets printed. Now we have real control! But be careful, if you try to use an index that doesn't exist, PHP gets angry:

```
// ...
$pets = array($pet1, $pet2, $pet3);

echo $pets[3];
```

> Notice: Undefined offset: 3 in /path/to/project/index.php on line 87

We'll talk more about array keys in the next chapter.

Putting other Stuff into an Array¶

We've only put strings into our array so far, but we can really put anything there, like a number:

```
<!-- index.php -->
<!-- ... -->

<?php
  $pet1 = 'Chew Barka';
  $pet2 = 'Spark Pug';
  $pet3 = 'Pico de Gato';

  $pets = array($pet1, $pet2, $pet3, 14);
?>
```

It's not a particularly exciting pet name, but when we refresh, we see "14" in our pet list.

An array is just a container that can hold anything. Each item in the array is given a unique key or index, which we can use to reference that item later if we need to. Heck, we can even put another array inside our array. We'll try that craziness next!

Arrays have a lot more power, alternate syntaxes and a load of functions that can operate on them. But right now, let's practice with the activities!

Chapter 4: Arrays Level 2: Associative Arrays

ARRAYS LEVEL 2: ASSOCIATIVE ARRAYS¶

Let's leave our array of pets behind and focus on one wonderful pet: Pancake the Bulldog! So far, we're just printing out the name of each pet, but Pancake has too much personality for that and deserves her own full bio! We'll print bios for every pet eventually, but let's just focus on Pancake first, because she's awesome!

I know a lot of information about Pancake, including her age, weight, bio and the filename to a photo. Let's store all of this information in an array and print it manually at the top of our pet list:

```
<!-- index.php -->
<!-- ... -->

<?php
  $pet1 = 'Chew Barka';
  $pet2 = 'Spark Pug';
  $pet3 = 'Pico de Gato';

  $pancake = array('Pancake the Bulldog', '1 year', 9, 'Lorem Ipsum', 'pancake.png');

  $pets = array($pet1, $pet2, $pet3);
?>

<div class="container">
  <div class="row">
    <div class="col-lg-4 pet-list-item">
      <h2><?php echo $pancake[0]; ?></h2>

      <img src="/images/<?php echo $pancake[4]; ?>" class="img-rounded">

      <blockquote class="pet-details">
        <?php echo $pancake[1]; ?>
        <?php echo $pancake[2]; ?> lbs
      </blockquote>

      <p>
        <?php echo $pancake[3]; ?>
      </p>
    </div>

    <!-- the foreach with $pets ... -->
  </div>
</div>
```

I'm using the array for convenience here, but eventually we'll query a database, which will give us an array of information just like this. When we refresh, we see Pancake displayed at the top of our list. And yes, we're weighing our dogs in pounds because we're from the US. Feel free to write kilograms if you like that better!

One big problem is that this is hard to read: it's not obvious that the 2 index is weight or that 4 is the image filename. And if we decided that we don't want to include weight anymore, Pancake's bio and filename keys change and the whole thing blows up!

> Notice: Undefined offset: 4 in /path/to/project/index.php on line 114

We could fix this, but the whole system is messy.

Specifying Array Keys¶

There's a better way of course! Instead of letting the array assign keys for us, let's tell the array exactly what key we want for

each item:

```
<!-- index.php -->
<!-- ... -->

<?php
   $pancake = array(
      'name' => 'Pancake the Bulldog',
      'age'  => '1 year',
      'weight' => 9,
      'bio' => 'Lorem Ipsum',
      'filename' => 'pancake.png'
   );
?>
```

First, I'll put each item on its own line. This is meaningless, I'm just trying to keep my code a bit more readable. When you want to control the key, put the key to the left of the item, followed by the equal and greater than signs. I sometimes call this an "equal arrow". Notice that these keys are strings, so we surround them by quotes. The end result looks like a map: the key on the left points to the value on the right.

Now all we need to do is update our code to use the keys. Again, now that they are strings, we surround them by quotes:

```
<!-- index.php -->
<!-- ... -->

<div class="container">
   <div class="row">
      <div class="col-lg-4 pet-list-item">
         <h2><?php echo $pancake['name']; ?></h2>

         <img src="/images/<?php echo $pancake['filename']; ?>" class="img-rounded">

         <blockquote class="pet-details">
            <?php echo $pancake['age']; ?>
            <?php echo $pancake['weight']; ?> lbs
         </blockquote>

         <p>
            <?php echo $pancake['bio']; ?>
         </p>
      </div>

      <!-- the foreach with $pets ... -->
   </div>
</div>
```

Refresh and success!

When you take control of the indexes, or keys, of an array, the array is known as an associative array. The name makes sense if you imagine associating each item in the array with a specific key. When an array is full of items where we *don't* specify the keys, it's known as a boring "indexed" array. I *may* have added the word boring.

associate: array('name' => 'Pancake', 'weight' => 9);

indexed: array('Pancake', 9);

Tip

Each item in an "indexed" still has an array key, but it's auto-assigned to a number, like 0, 1 or 2. We saw this in the last chapter.

Adding items to an Array after Creation¶

So far we're adding all the items to our array right when we create it. But how could we add more items to the array later?

Let's add a new `breed` to the array *after* it's been created:

```
<!-- index.php -->
<!-- ... -->

<?php
  // ...

  $pancake = array(
    'name' => 'Pancake the Bulldog',
    'age'  => '1 year',
    'weight' => 9,
    'bio' => 'Lorem Ipsum',
    'filename' => 'pancake.png'
  );

  $pancake['breed'] = 'Bulldog';

  $pets = array($pet1, $pet2, $pet3);
?>
```

Let's render it and refresh to make sure it works. Nice!

```
<!-- index.php -->
<!-- ... -->

<blockquote class="pet-details">
  <span class="label label-info"><?php echo $pancake['breed']; ?></span>
  <?php echo $pancake['age']; ?>
  <?php echo $pancake['weight']; ?> lbs
</blockquote>
```

Adding Items to an Indexed Array¶

While we're on the topic, can we also add more items to an indexed array after it's been created? Following what we did with the associative array, we could guess that it might look something like this:

```
<!-- index.php -->
<!-- ... -->

<?php
  $pet1 = 'Chew Barka';
  $pet2 = 'Spark Pug';
  $pet3 = 'Pico de Gato';

  // ... pancake code

  $pets = array($pet1, $pet2, $pet3);

  $pets[] = 'Kitty Gaga';
?>
```

But what key do we use between the square brackets? We could manually put in 3 (`$pets[3] = 'Kitty Gaga';`) since we can count the items in the array and see what the next key will be. But it would be better if PHP could automatically assign the key, just like it did for the other items.

To have PHP choose the index, we leave it exactly like this:

```
$pets[] = 'Kitty Gaga';
```

When you put nothing between the square brackets, it tells PHP to choose the key for us, which it does by picking the first available number (3 in this case).

In the next chapter, we're going to get crazy and use associative arrays to print more details on all of our pets. But first, let's practice!

Chapter 5: Arrays Level 3: We put Arrays in your Arrays!

ARRAYS LEVEL 3: WE PUT ARRAYS IN YOUR ARRAYS!¶

Ok, so we have associative arrays and indexed arrays. And really, they're the exact same thing: both contain items and each item has a unique key we can use to access it. We *choose* that key for items in an associated array and we let PHP choose the keys for us in an indexed array. And because PHP isn't very creative, it just chooses a number that gets higher each time we add something. But regardless of who makes the choice, every key in an array is either a string or a whole number, which we programmers and mathematicians call an `integer`. And that's the end of the story: array keys are *only ever* strings or integers in all of PHP.

```
   Key   => Value
array(
   'foo' => ?
   5     => ?
   'baz' => ?
);
```

But each *value* in an array can be *any* type of PHP value. So far we know three data types in PHP: a `string`, an *integer* and an `array`. And as promised, all three can be put into an array:

```
   Key   => Value
array(
   'foo' => 1500,
   5     => 'Hello World!',
   'baz' => array(1, 2, 3, 5, 8, 13),
);
```

Multi-dimensional Arrays¶

This means that we can have multi-dimensional arrays: an array with another one inside of it. Multidimensional arrays are actually pretty common and pretty easy. Let's tweak our code to make each pet an associate array, just like Pancake. I'll paste in the details:

```
$pet1 = array(
    'name' => 'Chew Barka',
    'breed' => 'Bichon',
    'age'  => '2 years',
    'weight' => 8,
    'bio'   => 'The park, The pool or the Playground - I love to go anywhere! I am really great at... SQUIRREL!',
    'filename' => 'pet1.png'
);

$pet2 = array(
    'name' => 'Spark Pug',
    'breed' => 'Pug',
    'age'  => '1.5 years',
    'weight' => 11,
    'bio'   => 'You want to go to the dog park in style? Then I am your pug!',
    'filename' => 'pet2.png'
);

$pet3 = array(
    'name' => 'Pico de Gato',
    'breed' => 'Bengal',
    'age'  => '5 years',
    'weight' => 9,
    'bio'   => 'Oh hai, if you do not have a can of salmon I am not interested.',
    'filename' => 'pet3.png'
);

$pancake = array(
    'name' => 'Pancake the Bulldog',
    'age'  => '1 year',
    'weight' => 9,
    'bio' => 'Lorem Ipsum',
    'filename' => 'pancake.png'
);
$pancake['breed'] = 'Bulldog';

// ...
```

Next, add `$pancake` to our `$pets` array and remove Kitty Gaga:

```
$pets = array($pet1, $pet2, $pet3, $pancake);
```

Notice that instead of passing a string as each item, we're now passing an array with lots of information about each pet. Before we go any further, let's use `var_dump` to see how this array looks. I'm also going to use a new function called `die`:

```
$pets = array($pet1, $pet2, $pet3, $pancake);
var_dump($pets);
die;
```

`die` kills the execution of the script immediately. It's useful for debugging because now our variable will print and `die` will temporarily prevent the rest of the page from rendering. That just makes things easier to read. In my development, `var_dump` and `die` go together like kittens and catnip. But as delicious as catnip is to a kitten, never use `die` in your real code.

When we refresh, we see the multi-dimensional array. Just like before, the outermost array is indexed with keys 0, 1 and 2. Each item is now an associative array with its own keys.

Accessing Data on a Multi-dimensional Array¶

So if we wanted to access the `breed` of the second pet in the list, how can we do that? It's actually wonderfully straightforward. First, access the second item by using the square bracket syntax, keeping in mind that array indexes start with 0. Next, add another set of square brackets with the breed key. Let's `var_dump` this to make sure it works:

```
$pets = array($pancake, $pet1, $pet2, $pet3);
$breed2 = $pets[1]['breed'];
var_dump($breed2);die;
```

Now that we have an array with details about multiple pets, we're dangerous! Look back at our `foreach` statement. We're still looping over `$pets`. But now, `$cutePet` is an associative array instead of a string:

```
foreach ($pets as $cutePet) {
  echo '<div class="col-lg-4">';
  echo '<h2>';
  echo $cutePet['name'];
  echo '</h2>';
}
```

In fact, we already did all this work when we rendered Pancake's details. Let's just re-use that code and change `$pancake` to `$cutePet`. I'll tweak a class name as well so that the our pets tile nicely.

```
<div class="row">
  <?php foreach ($pets as $cutePet) { ?>
    <div class="col-lg-4 pet-list-item">
      <h2><?php echo $cutePet['name']; ?></h2>

      <img src="/images/<?php echo $cutePet['filename']; ?>" class="img-rounded">

      <blockquote class="pet-details">
        <span class="label label-info"><?php echo $cutePet['breed']; ?></span>
        <?php echo $cutePet['age']; ?>
        <?php echo $cutePet['weight']; ?> lbs
      </blockquote>

      <p>
        <?php echo $cutePet['bio']; ?>
      </p>
    </div>
  <?php } ?>
</div>
```

Tip

I indented the `col-md-4` div 4 spaces inside the `foreach` just to help me read my code better - it doesn't change anything in PHP or HTML.

Refresh and voilà! To make things cleaner, I also close the PHP tag after my `foreach` statement. This lets me write HTML instead of printing it from inside PHP, which is hard to read. But it's really the same as before: we open PHP, start the `foreach`, close PHP, then later open it again to add the closing `}` for the `foreach`. If you're not used to this yet, we'll practice it!

Counting Items in an Array¶

So we're now doing *a lot* with arrays. Let's add one more thing! As cool as the `rand` function is, I want to print the real value for how many pets we have in the system. If there were a way to count the number of items in the `$pets` array, we'd be set. Fortunately, PHP gives us a function that does exactly that called `count`:

```
<!-- index.php -->
<!-- ... -->

<?php
  $cleverWelcomeMessage = 'All the love, none of the crap!';
  $pupCount = count($pets);
?>
```

When we refresh, we get an error:

> Notice: Undefined variable: pets in /path/to/index.php on line 70

The problem is that we're referencing the `$pets` variable, but it's not actually created until after this. PHP reads our file from top to bottom like a book, so we need to set a variable before using it.

To fix this, let's move every variable all the way up to the top of the file:

```
<!-- Right at the top of index.php -->
<?php
  $pet1 = array(
    'name' => 'Chew Barka',
    'breed' => 'Bichon',
    'age' => '2 years',
    'weight' => 8,
    'bio' => 'The park, The pool or the Playground - I love to go anywhere! I am really great at... SQUIRREL!',
    'filename' => 'pet1.png'
  );

  // .. the rest of the PHP code
  $pets = array($pet1, $pet2, $pet3, $pancake);

  $cleverWelcomeMessage = 'All the love, none of the crap!';
  $pupCount = count($pets);
?>
```

Now when we refresh, it works perfectly. If we add a 5th pet later, it will update automatically.

Let's go to php.net and look up the docs for the `count` function. As expected, it takes a single required argument. It also has a second, optional argument that you'll probably never use. You can tell it's optional because it's surrounded by square brackets. That's not really a PHP syntax, it's just a common way to document optional arguments.

While we're here, take a look at the left navigation: it's full of the functions in PHP that help you work with arrays. It's a massive list and has great stuff. For example, let's look at `array_reverse`. It takes an array as its one required argument, reverses it, and returns it. Let's use it to reverse `$pets`:

```
$pets = array($pancake, $pet1, $pet2, $pet3);
$pets = array_reverse($pets);
```

Sure enough, the pets reverse their order when we refresh. Notice also that I passed the `$pets` variable as the argument to `array_reverse` *and* set the result of the function to it. This is totally legal. The original value is passed to the function first and then the new, reversed value is set to `$pets` afterwards.

Congratulations on making it through this *tough* chapter. Now celebrate by dominating some exercises!

Chapter 6: Working with Files, JSON and Booleans

WORKING WITH FILES, JSON AND BOOLEANS¶

Right now our pet data array is just hardcoded and not dynamic. Eventually, we're going to pull this array from a database and let users add new pets to it. But before we get there, let's pretend someone has given us a file that contains all of adorable pets available on AirPupNMeow.com. Our goal will be to read that file and turn its contents into a PHP array that looks just like the one we're creating now by hand.

To make things easy, I've already prepared a file called `pets.json`, which I've put right inside my project directory.

Tip

You can find this file in the `resources/` directory of the code download.

Open up this file and see what's inside:

```
[
  {
    "name": "Chew Barka",
    "breed": "Bichon",
    "age": "2 years",
    "weight": 8,
    "bio": "The park, The pool or the Playground - I love to go anywhere! I am really great at... SQUIRREL!",
    "filename": "pet1.png"
  },
  {
    "name": "Spark Pug",
    "breed": "Pug",
    "age": "1.5 years",
    "weight": 11,
    "bio": "You want to go to the dog park in style? Then I am your pug!",
    "filename": "pet2.png"
  },
  {
    "name": "Pico de Gato",
    "breed": "Bengal",
    "age": "5 years",
    "weight": 9,
    "bio": "Oh hai, if you do not have a can of salmon I am not interested.",
    "filename": "pet3.png"
  },
  {
    "name": "Pancake",
    "age": "1 year",
    "weight": 9,
    "bio": "Treats and Snoozin!",
    "filename": "pancake.png",
    "breed": "Bulldog"
  }
]
```

Ok, let's step back and talk about JSON, which has nothing to do with PHP, except that PHP can read it. JSON is a text format that can be used to represent structured information, like pet details:

```
{
   "name": "Chew Barka",
   "breed": "Bichon",
   "age": "2 years",
   "weight": 8,
   "bio": "The park, The pool or the Playground - I love to go anywhere! I am really great at... SQUIRREL!","filename":"pet
}
```


In fact, any PHP array has a JSON string equivalent and we can turn PHP arrays into JSON and vice-versa. In fact, there's a function called `json_encode` which takes an array and returns the equivalent JSON string. Let's use it to see how our pet's array would look:

```
var_dump(json_encode($pets));die;
```

When we refresh, we see the JSON that's equivalent to our array. And in fact, this is exactly what we have in `pets.json`.

Tip

If you try this and your output looks a bit uglier, it's because I have a Chrome plugin called JSONView that adds spaces so that the JSON is very readable. Like with PHP, spaces don't make a difference in JSON. So, these two strings are equivalent, but the second is easier on the eyes!


```
{"name": "Chew Barka","breed":"Bichon"}
```



```
{
   "name": "Chew Barka",
   "breed": "Bichon"
}
```


The reason JSON exists is because squiggly braces are awesome! Or maybe it's so that different systems can communicate. Imagine if our website saved files that were sent off and read by some completely different application. JSON is magical because it can be read by PHP or any other language, like Ruby, Python or JavaScript. So even if that other application is built by a bunch of puppies, they'll be able to read our information. So, JSON is a great way to share data.

Back in PHP, let's pretend that there's already some other part of our site where users can submit new pets and that when they do, this `pets.json` file is being updated. Our job right now then is just to read its contents and display some pretty pet faces.

Reading and Opening Files¶

So first, how can we load the contents of a file in PHP? The answer is with the `file_get_contents` function. When we pull up its documentation, we can see how easy it is. Its only required argument is a filename. It opens up that file and returns its contents to us as a string.

Note

Remember that arguments surrounded by `[]` are optional. The optional arguments to `file_get_contents` are rarely used.

Easy! Let's use it and set the contents to a new variable! To see if it's working, we'll use our trusty `var_dump`:

```
$petsJson = file_get_contents('pets.json');
var_dump($petsJson);die;
```

When we refresh, we see the beautiful JSON string!

Warning and Errors in PHP¶

To experiment, let's change the filename and see what happens if the file doesn't exist:

```
$petsJson = file_get_contents('dinosaurs.json');
var_dump($petsJson);die;
```

this time, we see a warning from PHP:

> Warning: file_get_contents(dinosaurs.json): failed to open stream: No such file or directory in /path/to/index.php on line 16

PHP has both errors and warnings when things go wrong. The only difference is that if the code mistake isn't too bad, PHP just gives us a warning and tries to keep executing our code.

Tip

PHP also has notices, which act just like warnings.

Booleans: True and False¶

Here it continues, and executes our `var_dump`, which returns false. If we look at the documentation again, we see that `file_get_contents` returns the contents of the file as a string *or* it returns `false` if it couldn't read the file. `false` is called a Boolean, which is our fourth PHP data type. To review, we have:

1. Strings, like `$var = 'Hello World';`
2. Numbers, like `$var = 5;`. And actually, numbers are sub-divided into integers, like `5`, and floats, which have decimals like `5.12`. But most of the time in PHP, you don't care about this.
3. Arrays, like `$var = array('puppy1', 'puppy2', 4);`
4. And now our 4th type: Booleans. Booleans are simple because there are only two possible values: `true` and `false`:

```
$fileExists = false;
$iLikeKittens = true;
```

Like with the other 3 data types, we can assign Booleans to variables and functions can return Booleans. `file_get_contents` returns a string or `false`, which we now know is a `boolean` type.

Decoding JSON into an Array¶

Phew! Let's get back to our furry friends. First, fix the filename. Remember that the JSON contents we're reading from the file are a string and what we really want is to transform that JSON string into a PHP array. We used `json_encode` to turn an array into JSON, so it makes sense that we can use `json_decode` to go the other direction:

```
$petsJson = file_get_contents('pets.json');
$pets = json_decode($petsJson);
var_dump($pets);die;
```

When we refresh, it mostly looks right. But instead of an array, it says something about a "stdClass". This is a PHP object, which you don't need to worry about now. Instead, if we look at the `json_decode` docs, we see it has an optional second argument, which is a bool or Boolean that defaults to `false`. If we change this to `true`, the function should return an associative array:

```
$petsJson = file_get_contents('pets.json');
$pets = json_decode($petsJson, true);
var_dump($pets);die;
```

Perfect! This is the exact array we were building by hand, so remove that along with the `var_dump` statement:

```php
<?php
    $petsJson = file_get_contents('pets.json');
    $pets = json_decode($petsJson, true);

    // delete all the other $pet1 and $pets variables

    $pets = array_reverse($pets);

    $cleverWelcomeMessage = 'All the love, none of the crap!';
    $pupCount = count($pets);
?>
```

When we refresh, our page is back! The JSON string is read from the file and then converted into a PHP array. Our code is ready to iterate over each pet in that array and print out its information by using each pet's keys. This works because the information in the JSON file exactly matches the PHP array we had before.

If we changed the `filename` key for each pet in our data source `pets.json`, then we would also need to change it in our application to match:

```
[
    {
        "image": "pancake.png",
    },
]
```

```php
<!-- ... index.php -->

<?php foreach ($pets as $cutePet) { ?>
    <div class="col-md-4 pet-list-item">
        <!-- ... -->

        <img src="/images/<?php echo $cutePet['image']; ?>" class="img-rounded">

        <!-- ... -->
    </div>
<?php } ?>
```

Directory Path to a File¶

Refresh to make sure this still works. Before we finish, let's play with the PHP file-handling functions a little. First, move `pets.json` into a new directory called `data` and refresh. Oh no, things blow up!

> Warning: Invalid argument supplied for foreach() in /path/to/index.php on line 87

PHP no longer finds our file, which sets off a chain reaction of terrible things! First, `file_get_contents` returns `false`. Of course, `false` isn't a valid JSON string, so `json_decode` chokes as well and doesn't return an array like it normally would. Finally, we try to loop with `foreach`, but `$pets` isn't even an array. Woh! The moral is that sometimes a mistake in one spot will result in an error afterwards. So don't just look at the line number of the error: look at the lines above it as well.

To fix this, we can just change our file path to `data/pets.json`:

```php
$petsJson = file_get_contents('data/pets.json');
```

When we refresh, everyone is happy again! Notice that `file_get_contents` looks for files relative to the one being executed. We'll play with file paths more later, just don't think it's magic. PHP is happily stupid: it looks for files right in the directory of this one.

Note

You can also pass an absolute file path to PHP, like `/var/pets.json` or `C:\pets.json`.

Saving to a File¶

And what if you want to save data to a file? If we go back to the docs for `file_get_contents`, you'll see a related function: `file_put_contents`. It's also really simple: you give it a filename and a string, and it saves that string to that file. I'll let you try this on your own in the activities. Don't worry about its optional arguments.

Other ways to Read and Save Files¶

PHP has a bunch of other file-handling functions beyond `file_get_contents` and `file_put_contents`. These include `fopen`, `fread`, `fwrite` and `fclose`. For now, just forget these exist. Except for when you're dealing with very large files, these functions accomplish the exact same thing as `file_get_contents` and `file_put_contents`, they're just harder and weirder to use. To make matters worse, most tutorials on the web teach you to use these functions. Madness! You'll probably use them someday, but forget about them now. Working with files in PHP we need only our 2 handy functions.

Chapter 7: The wonderful if Statements

THE WONDERFUL IF STATEMENTS¶

Let's start to make our code smarter! Modify your `pets.json` file and remove the `age` key from just one of the pets. When we refresh, it doesn't fail nicely, it gives us a big ugly warning:

> Undefined index: age in /path/to/index.php on line 95

Let's dump the `$cutePet` variable inside the loop to see what's going on:

```
<?php foreach ($pets as $cutePet) { ?>
  <?php var_dump($cutePet); ?>
  ...
<?php } ?>
```

Each pet is an associative array, but as you probably suspected, Pico de Gato is missing her `age` key. When you reference a key on an array that doesn't exist, PHP will complain. Instead, let's code defensively. In other words, if we know that it's possible that the `age` key might be missing, we should check for it and only print the age if it's there.

To do this, we'll finally meet the wonderful and super-common `if` statement. Like *foreach*, it's a "language construct", and is one of those things that uses curly braces to surround a block of code:

```
<blockquote class="pet-details">
  <span class="label label-info"><?php echo $cutePet['breed']; ?></span>
  <?php
  if (true) {
    echo $cutePet['age'];
  }
  ?>
  <?php echo $cutePet['weight']; ?> lbs
</blockquote>
```

Where `foreach` accepts an array and executes the code between its curly braces one time for each item, `if` accepts a Boolean value - in other words `true` or `false`. If what you pass it is true, it executes the code between its curly braces.

In this case, I'm literally passing it the boolean `true`. The `echo` will always be called, since true will be true now, tomorrow and forever. What we really need is a function that can tell us if the `age` key exists on the `$cutePet` array.

That function is called `array_key_exists`. Let's look at its docs to make sure we know how it works. The first argument is the key, the second is the array, and it returns a Boolean. Perfect!

```
<blockquote class="pet-details">
  <span class="label label-info"><?php echo $cutePet['breed']; ?></span>
  <?php
  if (array_key_exists('age', $cutePet)) {
    echo $cutePet['age'];
  }
  ?>
  <?php echo $cutePet['weight']; ?> lbs
</blockquote>
```

Great! 3 pets have an age and one doesn't. This all happens with no warnings. `array_key_exists` returns true for 3 pets and false for our friend Pico de Gato.

If-else¶

But instead of rendering nothing if there is no age, let's print "Unknown". We can do this by adding an optional `else` part to our `if`:

```
if (array_key_exists('age', $cutePet)) {
    echo $cutePet['age'];
} else {
    echo 'Unknown';
}
```

When we refresh, it works! You'll use `if` statements all the time, both with and without the optional `else` part.

Combining If Conditions¶

Let's complicate things again by removing the age of another pet. But this time, don't remove the whole key, just set the age to an empty string.

```
{
    "name": "Chew Barka",
    "breed": "Bichon",
    "age": "",
    "weight": 8,
    "bio": "The park, The pool or the Playground - I love to go anywhere! I am really great at... SQUIRREL!",
    "image": "pet1.png"
}
```

When we refresh, we're still free of errors. But the age for Chew Barka is missing. Since it's blank, I would rather it say "Unknown".

If we dump the `$pets` array and refresh, we can see that this makes sense:

```
<?php var_dump($cutePet);die; ?>
<?php foreach ($pets as $cutePet) { ?>
    ...
<?php } ?>
```

Chew Barka has an `age` key, so `array_key_exists` returns true, and the age - which is a blank string - is printed out. What we really want is for the code in the `if` statement to only run if the `age` key exists *and* isn't blank.

Let's do this first by adding a new `if` statement inside our existing one. We'll check the age and only print it if it's *not* empty:

```
if (array_key_exists('age', $cutePet)) {
    if ($cutePet['age'] != '') {
        echo $cutePet['age'];
    }
} else {
    echo 'Unknown';
}
```

The `!=` is what you use when you want to compare 2 values to see if they are not the same. If the age is not empty, then this expression returns true and the echo runs.

Make sure also to add an `else` statement so that "Unknown" is printed if the `age` *is* empty:

```
if (array_key_exists('age', $cutePet)) {
    if ($cutePet['age'] != '') {
        echo $cutePet['age'];
    } else {
        echo 'Unknown';
    }
} else {
    echo 'Unknown';
}
```

This is all getting a little messy, but let's try it! When we refresh, 2 pets have ages, 2 say "Unknown", and we have exactly zero warnings. Nice!

The mess is that we have a lot of code for such a small problem. We also have the word "Unknown" written in 2 places. Code duplication is always a bummer because when you need to change this word later, you may forget about the duplication and only change it in one spot. Code duplication creates bugs!

Let's simplify. Really, we want to print the age if the `age` key exists *and* is not an empty string. We can just put both of these conditions in one `if` statement:

```
if (array_key_exists('age', $cutePet) && $cutePet['age'] != '') {
    echo $cutePet['age'];
} else {
    echo 'Unknown';
}
```

The secret is the double "and" sign, or ampersand if we are being formal. An `if` statement can have as many parts, or expressions in it as you want. This `if` statement has two expressions, the `array_key_exists` part and the part that checks to see if the age is empty. Each returns true or false on its own. By using `&&` between each expression, it means that every part must be true in order for the `if` statement to run. In other words, this is perfect.

Refreshing this time shows that things work just like before. But now our code is shorter, easier to read, and has no pesky duplication.

If-else-if¶

By now, you probably know that as soon as we get things working, I'll challenge us by adding something harder! Imagine that sometimes the dog owner knows the age of her dog, but purposefully wants to hide it. Let's change the age of Spark Pug to "hidden".

```
{
    "name": "Spark Pug",
    "breed": "Pug",
    "age": "hidden",
    "weight": 11,
    "bio": "You want to go to the dog park in style? Then I am your pug!",
    "image": "pet2.png"
}
```

When we see this, let's print a friendly message to contact the owner for the age.

We already have all the tools to make this happen, using another nested `if` statement:

```
if (array_key_exists('age', $cutePet) && $pet['age'] != '') {
    if ($cutePet['age'] == 'hidden') {
        echo '(contact owner for age)';
    } else {
        echo $cutePet['age'];
    }
} else {
    echo 'Unknown';
}
```

It works perfectly!

But let's see if we can flatten our code to use just one level of an `if` statement. There's nothing wrong with nested `if` statements, but sometimes they're harder to understand. We really have just 3 possible scenarios:

1. The `age` key does not exist or is blank. We print "Unknown".
2. The `age` key is equal to the string "hidden". For this, print our nice message about contacting the owner.
3. And if those other conditions don't apply, print the age!

When we had only one scenario, we just used an `if`. When we had two scenarios, we used an `if-else`. For 3 or more, we'll go crazy with an `if-elseif`:

```
if (condition #1) {
    echo 'Unknown';
} elseif (condition #2) {
    echo 'Hi! Email the owner for the age details please!';
} else {
    echo $cutePet['age'];
}
```

This is really how it looks, except for the "condition #1" and "condition #2" parts where we'll put real code that returns true or false. Like with the simple `if`, the `else` is optional, and you can actually have as many `elseif` parts as you want depending on how many different scenarios you have.

Tip

If you have many different scenarios, try using the somewhat rare, but handy switch case statement instead of a giant `if-elseif` block.

Combining Conditions with "or" and the not (!) Operator¶

Let's make our code follow this format. First, we need to check if the age key does not exist or if its value is empty. This is kind of the opposite of what we had before:

```
if (!array_key_exists('age', $cutePet) || $cutePet['age'] == '') {
    echo 'Unknown';
} elseif ('Condition #2') {

} else {

}
```

Ok, let's break this down. First, by putting the exclamation point in front of `array_key_exists`, it negates its value. If the function returns `true`, this changes it to `false` and vice-versa. We want the first part of our `if` to execute if the `age` key does *not* exist. The exclamation gives us that exactly.

Next, the `&&` becomes two "pipe" or line symbols (`||`). These mean "or" instead of and: we want our code to run if the `age` key does not exist *or* if its value is blank. Between `&&` and `||`, you can create some pretty complex logic in your `if` statements.

Tip

You can also use extra parenthesis to group conditions together, like you do in math. We'll see this later.

Finally, we used 2 equal signs (`==`) to see if the age value is equal to an empty string. This is *very* important: do not use a single equal sign when comparing 2 values. In fact, no matter where you are, repeat after me: "I solemnly swear to not use a single equal sign to compare values in an if statement".

The reason is that we use one equal sign to set a value on a variable:

```
// sets the age key to an empty string
$cutePet['age'] = '';
```

This is especially tricky because if you use only one equal sign the code will run. But, instead of comparing the two values, it sets the age to an empty string. This wouldn't break our code here, but it would in almost all other cases.

So when comparing values, use `!=` and `==`.

Tip

There are a few other symbols for comparing values, like `<` and `>` for comparing numbers. There is also a `===` symbol, which we'll talk about later. For a full list, see Comparison Operators

What is an Operator?¶

And by the way, these are called "operators". That's a generic word for a number of different symbols in PHP that operate on a value. We've seen a bunch so far, including `=`, which is called an assignment operator since it assigns a value to a variable. `&&` and `||` are called logical operators, they help combine different things to see if all of them put together are logically true or false. Knowing how to define an operator isn't important, just know that when you hear the word "operator", we're talking about some special symbol or group of symbols that do some special job.

Phew! Let's fill in the rest of our `if-elseif` statement, which should be pretty easy now:

```
if (!array_key_exists('age', $cutePet) || $cutePet['age'] == '') {
    echo 'Unknown';
} elseif ($cutePet['age'] == 'hidden')
    echo '(contact owner for age)';
} else {
    echo $cutePet['age'];
}
```

Try it! Oh man, a terrible error!

> Parse error: syntax error, unexpected 'else' (T_ELSE) in /path/to/index.php on line 101

Let's go to the line number and try to spot the problem. My editor helps me find it, but let's look ourselves. Always look first to see if you missed a semicolon - it's the most common mistake. And also look at the lines above the error. Ah ha! I forgot my opening `{` on the `elseif` part. Rookie mistake:

```
if (!array_key_exists('age', $cutePet) || $cutePet['age'] == '') {
    echo 'Unknown';
} elseif ($cutePet['age'] == 'hidden') {
    echo '(contact owner for age)';
} else {
    echo $cutePet['age'];
}
```

After fixing it, everything looks great.

Ok, you just learned a lot about if statements and using operators to compare values. I'll teach you some more tricks later, but now let's practice and get great with if statements.

Chapter 8: System Setup

SYSTEM SETUP¶

When you develop a website, it usually looks something like what you've been watching me do. You open up a PHP file in some sort of editor, then point your browser at some web address that executes the PHP file and shows you the output. But of course you've been practicing that diligently in the activities :). Now, it's time to get your computer setup so that you can program for realz!

Setting up your computer correctly is one of the hardest things you have to do... I mean really it sucks. It's a necessary evil and every computer is different, but we'll try to make this as easy as possible.

But first, we're going to learn a little bit about the invisible hamsters in wheels that run the web.

The Anatomy of Requesting a Page¶

Initially, there's you, on a browser, going to our domain - AirPup.com/index.php. When you hit enter, the magic starts. At this moment, our browser is making a "request", which means we're sending a message into the Interwebs. This message says that we're requesting the HTML for the `index.php` file on the AirPup.com domain. That file may be sitting on a completely different computer thousands of miles away–enjoying a beach you will never see. Eventually, our request message gets to that server, the `index.php` file is executed, and a "response" message full of HTML is sent back. Our browser gets that message and displays the HTML. Request, response, that's how the web works. It's like a game of telephone: you talk via your browser directly to a computer, tell it what you want, and it responds with it.

IP Addresses and the DNS Fairy¶

In order for the request to find its way to our server, a few magic things happen. First, a fairy called DNS turns AirPup.com into an IP address. There's more magic behind this, but an IP address is the address to a computer, just like you have a unique address to your house. With it, our request knows exactly which tubes to go through to find that server.

Web Server Software¶

When it gets there, it knocks on the door and says "I'm a web request, is there a website here?". Any machine that hosts a web site will have a piece of software running on it called a "web server". Don't get confused: the request has arrived at a server, which you should think of as a single machine, just like your computer. The "web server" is a just a program that runs all the time and waits for requests like ours to knock on the server's door.

Tip

There are many web servers available, but the most popular are Apache, Nginx and PHP's built-in web server.

At this point, the web server software opens the door and looks at our request message for AirPup's homepage. It says, "Yes, I do serve this website and its files live on this server at some specific location." That location is something you configure when you setup your web server. But let's pretend that the web server software is looking at `/var/www/airpup.com` for the AirPup files. Or if you're on Windows, perhaps it's looking at `C:/sites/airpup.com` or something like that. I'm just making these paths up - the important thing is that when the web server sees the request for AirPup.com, it knows that the files from this site live in a certain directory on the machine.

Requesting a File¶

Ok, just one more step! Let's pretend that this imaginary server out there on the web is actually my computer that you're watching right now. We usually think of servers living in huge rooms with blinking lights and the ability to withstand the zombie apocalypse, but my computer can also be a server. So when I type the domain into my browser, instead of going 1000 miles away across the web to some other machine, the request starts to go out and then comes back and knocks on the door to my computer. Let's also assume that I have some web server software running and it knows to look for the AirPup files right in this directory where I've been working.

So starting simple, if I change my URL to `http://AirPup.com/css/main.css`, I see my CSS file in the browser. The web server

software sees that we're requesting the file `/css/main.css` and so it opens up that file and sends a response to my browser with its contents. This looks simple, but we know that there are a bunch of players involved including my browser, the DNS fairy, the server, which happens to be my computer, and finally the web server software that grabs the file I've requested and "serves" it back to my browser. A web server is just something that serves files across the web. We send a message to request a file, it serves us a message back with that file's contents. Basically, this whole crazy system is setup to let other people access files on my computer.

Serving and Executing PHP Files¶

But when the file we're requesting ends in `.php`, something extra happens. Your web server software still finds the file like normal. When I put `index.php` in my browser, it finds the `index.php` in my directory. But instead of just returning that file unprocessed, with all of our PHP code, it executes that code. When all the code has been run, the final result is served as the response.

If we need another page, just create a new file called `contact.php`, put some simple code in there, and put `contact.php` in your address bar:

```
<?php echo 'Contact Us!'; ?>
```

Tip

In modern web development, we have a more robust way of handling multiple pages. You'll learn about that in an upcoming episode.

So what do you think will happen if we rename this file to `contact.html` and access it in the browser? Will the PHP code run? Will it just disappear?

When we try it, we get what *seems* like a blank page. But if you view the HTML source, there's our PHP code! Because the file didn't end in `.php`, our web server software simply returned the raw, unprocesssed file. The only reason we didn't see it at first is that the opening PHP tag looks like broken HTML, so our browser tries to process it.

Setting up Your Computer¶

Ok! Now it's your turn! If this seems a bit complex, don't worry. You already know more about how the web works than 99.9% of people that use it everyday. And there's a shortcut I'll show you to get all this setup easily.

Note

PHPTheRightWay.com has some additional notes on setting up your computer, including an alternative approach that uses a Virtual Machine. That's a great solution, but requires more setup than we want to cover here.

Install PHP¶

First, install PHP on your computer. PHP is basically just an executable file. Not an executable file like Chrome or iTunes that opens up and has a cute GUI, more like an executable that you can run from the command line. If you're not used to the command line, it's ok: we'll ease into it.

Since things seem to be most complex in Windows, I'll switch to Windows 7 for the rest of this chapter. Installing PHP is different in each system, so I'll have you follow along with me and some installation details for your operating system.

There are a lot of ways to install PHP, but the easiest is XAMPP, which works on Windows, Mac or Linux.

Note

If you're on a Mac and use MacPorts or Homebrew, you can install PHP through those. If you're on Linux and have a package manager like `apt-get` or `yum`, use it to install PHP.

Download XAMPP for your operating system. The PHP version doesn't matter, just get at least PHP 5.4. With the power of TV, I'll make our download look super fast. Now to follow along with the install instructions.

In addition to PHP, this also installs Apache - the most common web server software - and MySQL - the most common database. We won't worry about these right now.

To check if things are working, enter `http://localhost` in your browser and choose your language if it asks. You should see a

bright page. If you don't, don't panic. First, open up the XAMPP control panel and make sure Apache is running. If that's not the problem, ignore it for now. You may have already had Apache installed, which means they're fighting each other to answer the door. This is especially common on a Mac, which comes with a version of Apache and PHP already installed. We're not going to use Apache at all right now. So if your setup seems broken, ignore it!

Diagnosing how the XAMPP page works¶

By the way, how does this page work? In the address bar, instead of something like `airpup.com`, our domain is just `localhost`. If that seems odd to you because it has no `.com` or `.net` ending, good call! `localhost` is a special hostname that - on every machine in the world - is a pointer back to this same machine. So instead of going out to the DNS fairy and asking what the IP address of `localhost` is, the request immediately bounces right back to the computer you are using. `localhost` is a handy and very special shortcut for web developers.

Next, the request knocks on the door and Apache answers. Instead of pointing at a file, the URL in your browser may be just pointing at a directory called `xampp`. When we point to a directory, Apache is usually configured to look for an `index.php` file and render that. In fact, if we add `index.php` to the URL, nothing changes - this was the true file being displayed the whole time.

> http://localhost/xampp/index.php

Apache knows *where* on our computer the files of `localhost` live and looks for the `xampp/index.php` file there. So where is this directory? If we read the XAMPP docs, it's `C:\xampp\htdocs`. If we go there, we see a `xampp` subdirectory and an `index.php` file. Mystery solved. The directory where your server reads from is commonly called the `document root` or `web root` and its location will vary and can be configured.

Building our Project¶

Ok, enough with that! I *could* start building my project right inside the document root, but I'm going to put it somewhere else entirely, like a new `Sites` directory in my home directory. Create it if it's not there already.

Tip

Having a `Sites` folder in your home directory is a very common setup for Macs.

Apache doesn't ever look in here, so if I create an `index.php` file, it's not accessible via my web browser. If we wanted to use Apache, we'd need to reconfigure the document root to point here. But actually, we won't do that: I'll show you an easy trick instead.

But first, go to the KnpUniversity GitHub to download the code that goes along with this course. Choose the server_setup branch, download the zip file, unzip it into the Sites directory and rename it to `AirPup`. This is the code for the project we've built so far.

Using PHP's Web Server¶

Next, I'm going to turn Apache completely off. You don't need to do this, I just want to prove that we're not going to use it. Apache is great, but learning to use & configure Apache can bring its own headaches. I don't want us to worry about those right now.

When I refresh our page, it says that the server isn't found. Our request knocks on the door to our server, but since Apache is not running, no one answers and the request fails.

We *do* need a web server, but instead of using Apache, we're going to use PHP's itself. Since version 5.4, it has a built-in web server that's *really* easy to use. You won't use it on your real production server that hosts your finished website because it's not as fast as web servers like Apache. But for developing locally, it's wonderful.

First, open up a command line or terminal. Actually, XAMPP's control panel has a terminal we can use, which also sets up some variables and paths that make life easier. I'll use that.

Tip

In OS X, open up spotlight and type `terminal`. It's also in `Applications/Utilities/Terminal`.

Move into the directory you created. Of course, this looks different on Windows and your directory will live in a different location.

Tip

In OS X and Linux, if you created a `Sites/airpup` folder in your home directory, then you can move into it by typing `cd ~/Sites/airpup`.

Once here, type the following:

```
php -S localhost:8000
```

and hit enter. If your screen looks like mine, you're in luck! If you have an error or see something different, scroll down to the PHP Server Troubleshooting section in the script below to help you debug it.

Assuming it worked, just let this sit, copy the URL it printed, paste it into your browser, and add `index.php` to the end. Woh, it works! PHP is now our web server, and it looks right in this directory for its files.

To turn the server off, just press `Ctrl+c` or close the window. To turn it back on, run the command again. Don't forget to start this before you work.

Tip

By pressing "up", the terminal will re-display the last command you ran.

Port 8000 and Port 80¶

The `:8000` is called the port. A computer has many ports, which are like doors from the outside. By default, when a web request goes to a server, it knocks on port 80 and a web server, like Apache, is listening or watching that door. Most URLs don't have a `:80` on the end, only because your browser assumes that the request should be sent to port 80 unless you tell it otherwise. In our situation, we started the PHP web server and told it to listen on port 8000, not on port 80. There wasn't any special reason we did this and we could have listened on port 80 as well, as long as some other web server software weren't already watching that door. Because we did this, if a request goes to port 80, our PHP web server won't be there to answer. By adding `:8000`, the request goes to port 8000, our PHP web server is waiting, and everything continues like normal.

Congratulations! You have our PHP project running from your computer. You can start playing with the files to see what happens. Any editor can be used to edit the PHP files, since they're just plain text. But do yourself a favor and download a good editor: I recommend PHPStorm, NetBeans or Sublime Text if you have a Mac. PHPStorm will tell you when you have a syntax error, help you remember the arguments to PHP functions, and a lot more. It has a free trial so check it out. And no, they didn't even pay me to say that: they just have a great editor. But if they are listening... :).

PHP Server Troubleshooting¶

Everything working? Cool, you can skip this.

Having problem getting the built-in PHP web server running? Here are the most common problems:

Command php not Found¶

When you type the `php` command inside your terminal, you may get an error that basically says that the command `php` isn't found or recognized. PHP is an executable file that lives in a specific directory on your system. Normally, you can only execute files when you are actually *inside* the same directory as that file. But some files are registered in "global" paths. This means that when you type `php`, your computer knows exactly where on your system to find that file. If you've installed PHP and you're getting this error, this is the problem!

There are 2 fixes, and unfortunately they vary based on your system. First, to be safe, close and re-open your terminal. Sometimes this will fix things.

If it doesn't, here are the 2 options:

1. Read the Xampp documentation and find out where the PHP executable lives. Then, instead of typing simply `php`, you could type the full path to the executable file. This would mean typing something like:

   ```
   C:\xampp\php\php.exe -S localhost:8000
   ```

2. The above is a temporary fix, as it's pretty annoying to need to always include this full path. The real fix is to add the directory of your PHP executable to your system paths. In Windows, for example, there is a system configuration that says "when I type things at the command line, look in these directories for those files". If we can add the `C:\xampp\php` directory to that list, we're in luck! See how to access the command line for xampp on windows.

Running the command just returns a big set of directions¶

If you have an older version of PHP, then running `php -S localhost:8000` won't work. Instead, it will just print a big set of options back to you. If this happens, check your version of PHP by running `php -v`. If the version you have is 5.3.* or lower, you need to upgrade it. The command we're running in PHP only works for version 5.4.0 and higher. And that's really ok, 5.3 versions are quite old by now :).

Chapter 9: Creating Functions

CREATING FUNCTIONS¶

You've come a long way already, which has included using a bunch of built-in functions. Now it's time to make our own!

The first thing our PHP code does is read our `pets.json` file and decode it into an array, which we set on our `$pets` variable. Let's invent a new function called `get_pets` that will do this work for us and return the finished array. Custom functions are used just like core PHP functions, so eventually I want our code to look like this:

```
<?php
// $petsJson = file_get_contents('data/pets.json');
// $pets = json_decode($petsJson, true);
$pets = get_pets();

// ...
```

Oh, and those two slash marks (`//`) are one of the two ways you can comment out lines in PHP. Anything on a line after `//` is ignored by PHP entirely. This is handy for temporarily removing code or writing little love notes to your fellow developers.

Our new function can live anywhere in this file, but let's put it at the top and out of the way. Creating a function is really easy: just say `function`, give it a name, add a set of parenthesis, and finish with a set of opening and closing curly braces like we do with our 2 other special language constructs: `if` and `foreach`:

```
// index.php
// ...

<?php
function get_pets()
{

}

// $petsJson = file_get_contents('data/pets.json');
// ...
```

When we call this function, any code between the curly braces will be executed. Remember, the job of a function is usually to do some work and return a value. Eventually, we'll return an array of pets. But to get things working, just return a string by writing `return`, the string, then ending things with our usual semicolon:

```
// index.php
// ...

function get_pets()
{
    return 'woohoo';
}
```

To debug this, we'll of course `var_dump` the `$pets` variable, since it's set to the return value of our function. When we refresh, our string is dumped, proving that our function is working!

Now, just copy in the logic from before and make sure the function returns the array after decoding the string:

```php
// index.php
// ...

function get_pets()
{
    $petsJson = file_get_contents('data/pets.json');
    $pets = json_decode($petsJson, true);

    return $pets;
}
```

When we refresh, the whole system works! If you're wondering why this helped us, you'll see in a moment.

Chapter 10: Using require to Include Functions

USING REQUIRE TO INCLUDE FUNCTIONS¶

I'll show you a few more tricks with functions in a second. But right now, let's improve our `contact.php` page we started earlier. First, I want to be able to have a header that reads "Helping you find your new best friend from over 500 pets.":

```
<h1>
  Helping you find your new best friend from over 500 pets.
</h1>
```

Except, I want the number 500 to be dynamic, reflecting the true number of pets we have in our `pets.json` file. I'll do this by calling our `get_pets()` function and passing it to PHP's `count` function, which will tell us how many items are in that array:

```
<h1>
  Helping you find your new best friend from over <?php echo count(get_pets()); ?> pets.
</h1>
```

But when we try it, it blow up big time!

> Fatal error: Call to undefined function get_pets() in /php1/contact.php on line 2

Hmm, the function works in `index.php`, but not in `contact.php`. Clearly, the difference is that `get_pets()` actually lives in `index.php`. And that's totally correct. Even though it sits next to `index.php`, when PHP executes the contact page, it has no idea what functions or variables might live in the index file. These two files exist in perfect isolation.

So is there a way to load the functions in `index.php` from `contact.php`? Of course! The answer is with the `require` statement, which tells PHP to load & parse the contents of some other file. Require `index.php` at the top of the file:

```
<?php
// contact.php
require 'index.php';
?>

<h1>
  Helping you find your new best friend from over <?php echo count(get_pets()); ?> pets.
</h1>
```

When we try it, it works... sort of. At the bottom of the page, we see our sentence with the correct pets count. But above it is the entire `index.php` page, which we did *not* want. Adding `require` made using the `get_pets()` function possible, but it also brought in all the HTML from the index page as well. What can we do?

Let's create a new file called `functions.php`, which I'll put in a `lib/` directory for organization since this PHP file isn't meant to be a page that's accessed directly like our index and contact files. Now, move the `get_pets()` function in here, being sure to remember your opening PHP tag:

```php
<?php
// lib/functions.php

function get_pets()
{
    $petsJson = file_get_contents('data/pets.json');
    $pets = json_decode($petsJson, true);

    return $pets;
}
```

No Closing PHP Tag?¶

If you're screaming that I forgot the closing PHP tag, you're half-right. If the last thing you have in a file is PHP code, adding the closing PHP tag is optional, and it's actually better if you leave it off. If that confuses you, go ahead and close your PHP tags for now.

Tip

Why is *not* closing your PHP tags better when you don't have to? Great question - see http://bit.ly/IsCwPE.

Using functions.php¶

Now, simply `require` `lib/functions.php` from both the index and contact files:

```php
<?php
// index.php
require 'lib/functions.php';
// ...
```

```php
<?php
// contact.php
require 'lib/functions.php';
// ...
```

Both will now have access to the `get_pets` function, but without any extra HTML, since `functions.php` doesn't have any. When we try them, they both work!

require, require_once, include, include_once¶

Actually, there are 4 statements that can be used to execute an external file:

- `require` ;
- `require_once` ;
- `include` ;
- `include_once` .

The difference between `require` and `require_once` is simple: `require` will *always* load a file while `require_once` will make sure that it only loads the file *one* time no matter how many times you call it. So if `functions.php` were doing some initialization and it was *really* important to only include this file once, we might use `require_once` just to be safe. This becomes more important later when you work with classes.

The other two statements are `include` and `include_once` . These are exactly the same as `require` and `require_once` , except that if the file doesn't exist, include let's the script keep running. On the other hand, if a file imported with `require` is missing, a fatal error will occur and your page will be killed immediately. In practice, I almost never use `include` , because I have a hard time imagining a scenario where my app is including another PHP file that only *might* exist. Does it take a break every 15 minutes and leave my server?

To keep things simple, use `require` or `require_once` , if you need to. But realize two things. First, all 4 of these do the same thing. And second, if you stick with us, you'll be programming sites that are so well-built that you will practically stop using

any of these. But, I’m getting ahead of myself.

Chapter 11: Adding a very Simple Layout

ADDING A VERY SIMPLE LAYOUT¶

Ok, let's improve one more thing by giving `contact.php` the same HTML layout as our homepage. To do this, just isolate the header and footer into their own files, say `layout/header.php` and `layout/footer.php`. Move all of the HTML from the top of the index page into header and the stuff from the bottom into footer:

```
<!-- layout/header.php -->
<!DOCTYPE html>
<html lang="en">
<head>...</head>
<body>
<!-- header code here... -->
```

```
<!-- layout/footer.php -->
<footer>...</footer>
</body>
</html>
```

Require the header near the top of each page and the footer after our page-specific code:

```
<?php
    // index.php
    // ... the big PHP code block at the top
?>
<?php require 'layout/header.php'; ?>

<!-- all of the HTML and PHP code from the middle  ...-->

<?php require 'layout/footer.php'; ?>
```

Note

Repeat the same thing in the contact.php file.

And just like that, our two pages are sharing the layout!

You're going Far!¶

Woh! You just learned a ton: coming from the basics of PHP to using arrays, functions, foreach loops, complex if statements, true/false boolean values, user-defined functions, require statements and more. We also got your system setup so that you can develop and try your own code locally.

So congratulations, but keep going! There's so much more that we need to cover, like how to submit a form, talk to a database, and use query parameters. And to be serious developers, we also need to organize our project better. As nice as the `functions.php`, `header.php`, and `footer.php` files are, real developers organize things in more sophisticated ways. But keep working and we'll get there!

See ya next time!