**Layout with Flex**

We've already looked at **display:inline-block** and **vertical-align:top** to change our layouts, but there is another more modern method known as **Flexbox.**

Unlike inline-block, flex is a system that works by defining parent elements as **flex containers.**These flex containers turn all their children into **flex items.**

Let's look at a header that has two items in it:

A screenshot of a social media post

Description automatically generated

There is a logo and a search bar. Say we want the search bar to be at the opposite end if the header. Flex makes this easy! Let's target the header in CSS and change it to a flex container:

A screenshot of a social media post

Description automatically generated

Now header's two children, img and form, are next to each other. Because header is a flex container now, we can use other flex features to create the layout we want. One of them is **justify-content:space-between.**

A screenshot of a social media post

Description automatically generated

No matter how big our browser is, our items will now be perfectly separated. Nice! What if we wanted to center them vertically? We can use **align-items:center.**

A screenshot of a social media post

Description automatically generated

Excellent! With only 3 CSS properties we now have a nice header bar.

Try playing with the different values for **justify-content**and **align-items.**

Another great flex property is **flex-wrap:wrap;**. Setting this on your flex container will automatically adjust your flex items to a single column when the browser is too small. Mobile-friendly!

# Browser Compatibility

Different browsers may give default attributes to HTML elements, causing sites to look different based on the version of the browser or whether you're seeing the site in IE, Firefox, Chrome, etc.  For example in Internet Explorer, an H1 element may have certain margin and padding causing your site to look one way, while Chrome and Firefox may give the H1 a slightly different margin/padding.  These differences can cause a headache, especially if you're trying to make your site look consistent throughout different browsers/versions.

To solve many cross-browser issues, it's encouraged that you use 'RESET' and 'NORMALIZE' (discussed below) and, basically, define what the default attributes should be (instead of relying on the default values that come with the browser).  In fact, **technical recruiters love to ask you about this** and see if you know how to make your site look consistent throughout different browsers.  When they ask you this, you should be able to explain (with confidence) what reset/normalizing means and why they are used.  You should also tell them that in addition to this, it's important to validate your HTML/CSS often, as validating the HTML/CSS gives insight on how you can fix issues that may lead to cross-browser compatibility issues (and often, invalid HTML/CSS can lead to weird behavior as the browser may not 'fix' invalid HTML/CSS quite the right way).

## CSS Reset

Here is a great explanation of what CSS resets are, why we use them, and some examples:

<http://perishablepress.com/a-killer-collection-of-global-css-reset-styles/>

## Normalize

A widely used alternative to CSS resets is normalize.css.  Normalize.css also covers HTML5 elements. Find more information about what it does and why it might be preferable to CSS resets here:

<https://github.com/necolas/normalize.css/>

**FAQ**

**Q: Can I add multiple classes to an element?**

**A: Yes.** Simply separate the class names with space. Ex:

<div class="nav side-bar"></div>

**Q: Can I add a class and an id to the same element?**

**A: Yes.**An element can have an id and multiple classes. Ex:

<div id="primary-nav" class="nav side-bar"></div>

**Q: Do I have to add a class or an id to target a nested element?**

**A: Not as a rule.**Consider the following example:

<ul id="nav-menu">

<li><a href="#">Home</a></li>

<li><a href="#">Dashboard</a></li>

<li><a href="#">About</a></li>

<li><a href="#">Contact Us</a></li>

</ul>

To target the links and remove the underlines from them we can do:

#nav-menu li a{

text-decoration: none;

}

This tells the browser to first locate the element with id nav-menu, then find all the list items inside the unordered lists, then find the links inside those list elements.

**In general, make your code easier to read by avoiding adding too many classes and ids.**

# Bootstrap

## Objectives

1. Get familiar with Bootstrap, a very popular css library/framework used by lots of developers
2. As developers, you'll most likely use an existing CSS file created by others.  This is a good opportunity for you to know how it's like to leverage another css framework that's created by other folks.
3. For you to appreciate the simple design that Bootstrap provides.

## Introduction

### What is Bootstrap?

In the words of its creators, bootstrap is a "sleek, intuitive, and powerful mobile-first front-end framework for faster and easier web development."

To get the most up to date version of Bootstrap use [this](https://www.bootstrapcdn.com/) link.  An example is shown below.

To use Bootstrap, first put the following lines in the <head> section of your file:

<head>

<!-- Required meta tags -->

<meta charset="utf-8">

<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

<!-- load Bootstrap -->

<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css">

<!-- Optional JavaScript -->

<!-- jQuery first, then Popper.js, then Bootstrap JS -->

<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js" integrity="sha384-q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo" crossorigin="anonymous"></script>

<script src="://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js" integrity="sha384-UO2eT0CpHqdSJQ6hJty5KVphtPhzWj9WO1clHTMGa3JDZwrnQq4sF86dIHNDz0W1" crossorigin="anonymous"></script>

<script src="://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js" integrity="sha384-JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEaFf/nJGzIxFDsf4x0xIM+B07jRM" crossorigin="anonymous"></script>

</head>

Now you can go ahead and add Bootstrap classes to your HTML elements in order to easily achieve consistent, clean, and responsive design.

# Page Layout

## Using Bootstrap for Layouts

For starters, we are going to look at general layout using containers, columns and the grid system.  Bootstrap utilizes flexbox to develop a grid system where a container is divided up into 12 even columns.  A .container class is a required parent in order to use columns.  Consider the following code:

<div class="container">

    <div class="row">

        <div class="col"></div>

    <div class="col"></div>

    <div class="col"></div>

    </div>

</div>

This will produce 3 evenly spaced columns on the page, 4+4+4=12.  If we add more bootstrap classes to the code, we would be able to see the results seen below.

<div class="container text-center">

    <div class="row">

        <div class="col bg-secondary text-light m-2">One</div>

        <div class="col bg-secondary text-light m-2">Two</div>

        <div class="col bg-secondary text-light m-2">Three</div>

    </div>

</div>



Columns can also be nested inside of other columns.  In this example the nested column will be divided into 3 even columns of the parent column.

<div class="container text-center">

    <div class="row">

        <div class="col bg-secondary text-light m-2">One</div>

        <div class="col bg-secondary text-light m-2">Two

            <div class="row">

                <div class="col bg-info text-light m-2">A</div>

                <div class="col bg-info text-light m-2">B</div>

                <div class="col bg-info text-light m-2">C</div>

            </div>

        </div>

        <div class="col bg-secondary text-light m-2">Three</div>

    </div>

</div>



We can even set how wide we want the column to be, and offset the width of the column in the layout.  The below example will center the column in the page and have a default width of 6 of the available column grids.

<div class="container border border-dark">

    <div class="row">

        <div class="col-6 offset-3 bg-secondary text-light">Center column, half of the viewport width</div>

    </div>

</div>

#### Useful classes for layouts

Containers - [https://getbootstrap.com/docs/4.4/layout/overview/](https://getbootstrap.com/docs/4.0/layout/overview/) - (just learn about "container" class and skip everything else)

Grid system - <https://getbootstrap.com/docs/4.4/layout/grid/> - (for positioning and sizing of columns):

Color schema - <https://getbootstrap.com/docs/4.4/utilities/colors/> - (helpful to see the columns being created)

Spacing - <https://getbootstrap.com/docs/4.4/utilities/spacing/> - (paddings and margins)

Sizing - <https://getbootstrap.com/docs/4.4/utilities/sizing/> - (height and width of elements)

e hope you get the point.  **Putting only one element in another element is like putting your Christmas gift in a box and then putting that box in another box that's slightly bigger (which means that the outer box was not necessary).**

**Great programmers always do things with less code**, and we want you to practice reviewing your code and always striving to do things in the most simple manner possible. Fewer lines of code -- less text -- this is always better. Please remember that.

Now, duplicate the page below while considering what you've learned above.

A screenshot of a cell phone

Description automatically generated

* It's recommended that you reset and normalize the CSS attributes before you add your own CSS attributes.
* Download the [cropped images](http://s3.amazonaws.com/General_V88/boomyeah/company_209/chapter_3921/handouts/chapter3921_7219_www-images.zip) to be used for this exercise.
* Make sure you've validated your HTML and that you're not using any float: left or right.
* Make sure you double checked your code for any redundant use of ids or classes. Make sure you don't have any single element within a division.
* Refactor your HTML code, if necessary.

# Functions

## Objectives:

* Understand what a function is and why it's important
* Understand the basic syntax for a Python function
* Understand parameters and arguments
* Understand what a return does in a function

A **function** is a **named block of code that we can execute to perform a specific task**. More simply, a function is a list of instructions that we can run at any time and as many times as we would like. If we find something that we seem to be using over and over again, it might be best to have a way to streamline the process. A function:

* has a name
* takes in parameters (parenthesis required, parameters optional)
* perform a series of instructions
* return something afterwards (will return None if there is no explicit return statement)

Think of the function as a factory. If we were building a new car we would:

1. Specify the features (variables) needed for creating a car
2. Send the specific features (pass arguments) to a car manufacturing plant (invoke a function)
3. The factory (function) receives the specifications (parameters) and does something with them
4. The factory sends a car (return) back to us, since we sent in the request

A picture containing drawing

Description automatically generated

The factory has all the instructions to build a new car and will perform all the tasks. When you want a new car, all you have to do is call the factory to request a new car.

The advantages of using functions are:

* Reducing the duplication of code
* Breaking down complex problems into simpler pieces
* Improving clarity of code

## Syntax

The def keyword signifies the declaration of a function. This indicates that the following code is a function and assigns a name to that function, so we can **call** it later. Parameters are **inputs** the function is expecting and appear inside the parenthesis that follow the function name.

Here's a basic example of a function:

*def* add(a,b): # function name: 'add', parameters: a and b

x = a + b # process

return x # return value: x

We have declared a function with the def keyword, named it add, and specified that it takes two inputs (parameters). If this is all we have in our file, nothing would actually appear to happen if we ran it. To actually run the function, we must execute it by **invoking** or **calling** it. This is done outside of the function using the function name followed by **()**. Inside the parenthesis are any values (arguments) the function is expecting as input.

new\_val = add(3, 5) # calling the add function, with arguments 3 and 5

print(new\_val) # the result of the add function gets sent back to and saved into new\_val, so we will see 8

Once invoked, a function can give us an output. Some functions take an input and some functions don't give us an output. Even if no output is produced, the code inside the function can alter the program - this is called a side effect. Based upon what we learned above, a function that doesn't return anything would produce no output!

## Parameters and Arguments

We define the input of functions using **parameters.** Functions can have as many parameters as we need, including 0. Here we've defined the say\_hi function with one parameter called name:

*def* say\_hi(name):

print("Hi, " + name)

Now, we can invoke this function by calling its name and passing in the correct number of arguments:

# invoking the function 3 times, passing in one argument each time

say\_hi('Michael')

say\_hi('Anna')

say\_hi('Eli')

Wait, but what's the difference between a parameter and an argument? These two words get mixed up a lot in programming. In this example **'name' is a parameter** while **"Michael", "Anna", and "Eli", are arguments**. We **define parameters.** We **pass in arguments** into functions.

Here's the output:

A screenshot of a cell phone

Description automatically generated

## Returning Values

So far none of our functions had any value that we could hold onto. In many cases, we would want our function to **return** some sort of value that we can use later in our program. The following concept is critical in understanding how to use functions correctly in your code:

It is very important to remember the following: **a function call is equal to whatever that function returns**. This might not make sense until we see it in action.

Let's modify our original say\_hi function and observe the differences:

*def* say\_hi(name):

return "Hi " + name

greeting = say\_hi("Michael") # the returned value from say\_hi function gets assigned to the 'greeting' variable

print(greeting) # this will output 'Hi Michael'

Returning a value from a function allows us to store that value in a variable. In this example, we invoked the say\_hi function with "Michael" and set it to the greeting variable. When we print greeting we see that it contains the returned value of the say\_hi function - "Hi Michael".

Going back to our add function, recall that it takes two parameters and returns the sum of the parameters.

*def* add(a, b):

x = a + b

return x

sum1 = add(4,6)

sum2 = add(1,4)

sum3 = sum1 + sum2

What do you think the values of sum1, sum2, and sum3 would be?

If you guessed 10, 5, and 15, respectively, good job! sum1 was set to the return value of the add function invoked with 4 and 6 as arguments. Similarly, sum2 was set to the return value of invoking add with 1 and 4. The variable sum3 contains the sum of sum1 and sum2 which is 15. Storing these return values in variables allows us to use the results of our functions throughout the rest of our program.

In our examples you may have noticed that our functions were returning values of different data types. Functions can return any of the data types - strings, numbers, lists, tuples, dictionaries and even other functions!