Flexbox:

- **Flexbox** is a layout model that allows elements to align and distribute space within a container. Using flexible widths and heights, elements can be aligned to fill a space or distribute space between elements, which makes it a great tool to use for responsive design systems.
- Oftentimes we want to align elements on a page side by side as opposed to just vertically down. This is a bit of a challenge at times without an effective strategy to help us. The **flexbox strategy** is a method of aligning and distributing items in a container even if their size is unknown and/or dynamic. It is one of several strategies of laying out div content on an HTML page. Another popular one is the grid strategy.
- The main idea behind the flex layout is to give the container the ability to change its items' width, height, and order in order to fill the available space. A flex container expands items to fill the remaining free space or conversely shrink them to prevent them from overflowing from the container itself.
- From now on, when you plan to align items, you will use the following strategy:
    - Create a flex container.
    - Add properties to the flex container to satisfy your alignment requirements.
    - Create flex items inside the container.
- The flexbox strategy makes use of two components:
    1. **Flex container:** The div surrounding the items to be aligned.
    2. **Flex items:** The items to be aligned.
- To setup a flexbox, we can set up a container div with the CSS property flex as follows:
```
.container {
   display: flex;
}
```
- Some properties of flex container are:
    1. **flex-direction:**
        - The flex-direction property defines in which direction the container wants to stack the flex items.
        - The row value stacks the flex items horizontally from left to right.
        - The row-reverse value stacks the flex items horizontally but from right to left.
        - The column value stacks the flex items vertically from top to bottom.
        - The column-reverse value stacks the flex items vertically but from bottom to top.
        - E.g.
```
.container {
   flex-direction: row;            /* default order (left to right) */
   flex-direction: row-reverse;    /* right to left order */
   flex-direction: column;         /* top to bottom order */
   flex-direction: column-reverse; /* bottom to top order */
}
```
        - Example with flex-direction row:
```
.flex-container {
  display: flex;
  flex-direction: row;
}
```

- Example with flex-direction row-reverse:
  ```
  .flex-container {
    display: flex;
    flex-direction: row-reverse;
  }
  ```
- Example with flex-direction column:
  ```
  .flex-container {
    display: flex;
    flex-direction: column;
  }
  ```
- Example with flex-direction column-reverse:
  ```
  .flex-container {
    display: flex;
    flex-direction: column-reverse;
  }
  ```

2. flex-wrap:

- The flex-wrap property specifies that the flex items will wrap if necessary.
- The wrap value specifies that the flex items will wrap if necessary.
- The nowrap value specifies that the flex items will not wrap. This is default.
- The wrap-reverse value specifies that the flexible items will wrap if necessary, in reverse order.
- E.g.
  ```
  .container {
     flex-wrap: wrap;     /* wrap onto multiple lines from top to bottom */
     flex-wrap: nowrap;  /* default (all items on the same line */
     /* wrap onto multiple lines from bottom to top */
     flex-wrap: wrap-reverse;
  }
  ```
- Example with wrap:
  ```
  .flex-container {
    display: flex;
    flex-wrap: wrap;
  }
  ```
- Example with nowrap:
  ```
  .flex-container {
    display: flex;
    flex-wrap: nowrap;
  }
  ```
- Example with wrap-reverse:
  ```
  .flex-container {
    display: flex;
    flex-wrap: wrap-reverse;
  }
  ```

3. flex-flow:

- The flex-flow property is a shorthand property for setting both the flex-direction and flex-wrap properties.

- Example with flex-flow:
```
.flex-container {
  display: flex;
  flex-flow: row wrap;
}
```

4. justify-content:

- The justify-content property is used to align the flex items horizontally.
- It defines the alignment along the main axis. It helps distribute extra free space leftover when either all the flex items on a line are inflexible, or are flexible but have reached their maximum size. It also exerts some control over the alignment of items when they overflow the line.
- The center value aligns the flex items at the center of the container.
- E.g.
```
.flex-container {
  display: flex;
  justify-content: center;
}
```
- The flex-start value aligns the flex items at the beginning of the container. This is default.
- E.g.
```
.flex-container {
  display: flex;
  justify-content: flex-start;
}
```
- The flex-end value aligns the flex items at the end of the container.
- E.g.
```
.flex-container {
  display: flex;
  justify-content: flex-end;
}
```
- The space-around value displays the flex items with space before, between, and after the lines.
- E.g.
```
.flex-container {
  display: flex;
  justify-content: space-around;
}
```
- The space-between value displays the flex items with space between the lines.
- E.g.
```
.flex-container {
  display: flex;
  justify-content: space-between;
}
```

5. align-items:

- The align-items property is used to align the flex items vertically.
- The align-items property defines the default behavior for how items are laid out along the cross axis, which is perpendicular to the main axis.

- You can think of align-items as the justify-content version for the cross-axis.
- The center value aligns the flex items in the middle of the container.
- E.g.
```
.flex-container {
  display: flex;
  height: 200px;
  align-items: center;
}
```
- The flex-start value aligns the flex items at the top of the container.
- E.g.
```
.flex-container {
  display: flex;
  height: 200px;
  align-items: flex-start;
}
```
- The flex-end value aligns the flex items at the bottom of the container.
- E.g.
```
.flex-container {
  display: flex;
  height: 200px;
  align-items: flex-end;
}
```
- The stretch value stretches the flex items to fill the container. This is default.
- E.g.
```
.flex-container {
  display: flex;
  height: 200px;
  align-items: stretch;
}
```
- The baseline value aligns the flex items such as their baselines aligns.
- E.g.
```
.flex-container {
  display: flex;
  height: 200px;
  align-items: baseline;
}
```

6. **align-content:**
    - The align-content property is used to align the flex lines.
    - It helps to align a flex container's lines within it when there is extra space in the cross-axis, similar to how justify-content aligns individual items within the main-axis.
    - **Note:** This property has no effect when the flexbox has only a single line.
    - The space-between value displays the flex lines with equal space between them.

- E.g.
```
.flex-container {
  display: flex;
  height: 600px;
  flex-wrap: wrap;
  align-content: space-between;
}
```
- The space-around value displays the flex lines with space before, between, and after them.
- E.g.
```
.flex-container {
  display: flex;
  height: 600px;
  flex-wrap: wrap;
  align-content: space-around;
}
```
- The stretch value stretches the flex lines to take up the remaining space. This is default.
- E.g.
```
.flex-container {
  display: flex;
  height: 600px;
  flex-wrap: wrap;
  align-content: stretch;
}
```
- The center value displays the flex lines in the middle of the container.
- E.g.
```
.flex-container {
  display: flex;
  height: 600px;
  flex-wrap: wrap;
  align-content: center;
}
```
- The flex-start value displays the flex lines at the start of the container.
- E.g.
```
.flex-container {
  display: flex;
  height: 600px;
  flex-wrap: wrap;
  align-content: flex-start;
}
```
- The flex-end value displays the flex lines at the end of the container.
  E.g.
```
.flex-container {
  display: flex;
  height: 600px;
  flex-wrap: wrap;
  align-content: flex-end;}
```

- Some properties of flex item are:
    1. **order:**
        - The order property specifies the order of the flex items. By default, they are ordered in the order in which they appear.
        - The order value must be a number. The default value is 0.
        - E.g.

```
<div class="flex-container">
  <div style="order: 3">1</div>
  <div style="order: 2">2</div>
  <div style="order: 4">3</div>
  <div style="order: 1">4</div>
</div>
```

    2. **flex-grow:**
        - The flex-grow property specifies how much a flex item will grow relative to the rest of the flex items.
        - By default, all items have flex-grow set to 0.
        - The value must be a number. The default value is 0.
        - E.g. This makes the third flex item grow eight times faster than the other flex items.

```
<div class="flex-container">
  <div style="flex-grow: 1">1</div>
  <div style="flex-grow: 1">2</div>
  <div style="flex-grow: 8">3</div>
</div>
```

    3. **flex-shrink:**
        - The flex-shrink property specifies how much a flex item will shrink relative to the rest of the flex items.
        - The value must be a number. The default value is 1.
        - E.g. This does not let the third flex item shrink as much as the other flex items.

```
<div class="flex-container">
  <div>1</div>
  <div>2</div>
  <div style="flex-shrink: 0">3</div>
  <div>4</div>
  <div>5</div>
  <div>6</div>
  <div>7</div>
  <div>8</div>
  <div>9</div>
  <div>10</div>
</div>
```

    4. **flex-basis:**
        - The flex-basis property specifies the initial length of a flex item.
        - E.g. This sets the initial length of the third flex item to 200 pixels.

```
<div class="flex-container">
  <div>1</div>
  <div>2</div>
  <div style="flex-basis: 200px">3</div>
  <div>4</div>
</div>
```

5. **flex:**
    - The flex property is a shorthand property for the flex-grow, flex-shrink, and flex-basis properties.
    - E.g. This makes the third flex item not growable (0), not shrinkable (0), and with an initial length of 200 pixels.

```
<div class="flex-container">
  <div>1</div>
  <div>2</div>
  <div style="flex: 0 0 200px">3</div>
  <div>4</div>
</div>
```

6. **align-self:**
    - The align-self property specifies the alignment for the selected item inside the flexible container.
    - The align-self property overrides the default alignment set by the container's align-items property.
    - E.g. This aligns the third flex item in the middle of the container.

```
<div class="flex-container">
  <div>1</div>
  <div style="align-self: flex-start">2</div>
  <div style="align-self: flex-end">3</div>
  <div>4</div>
</div>
```

JavaScript:

- Javascript is an object oriented programming language that is used to make web pages interactive.
- It's an interpreted language and runs on the client's computer/browser.
- With respect to the other 3 web-development languages:
    1. HTML defines the content of a web page. It defines the information that we see on the page.
    2. CSS defines the style of a web page. It defines what certain things are supposed to look like.
    3. Javascript defines the functionality of a web page. It defines what happens when we click on things or input data.
- Javascript can be included in HTML code using the <script></script> tag. Similarly to CSS, it can be included using either inline Javascript or using external Javascript.
- Here's an example of inline Javascript:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>My Website</title>
</head>
<body>
  <p>This is some content</p>
  <script>
```

```
    // we can write our javascript code here as we wish
    console.log('keviniscool');
  </script>
</body>
</html>
```

- Here's an example of external javascript:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>My Website</title>
</head>
<body>
  <p>This is some content</p>
  <script src="script.js"></script>
</body>
</html>
```

- There are two main differences with including Javascript when compared to including CSS:
    1. We include both inline and external Javascript using the <script></script> tag.
    2. Javascript is included at the bottom of the file as opposed to the <head> tag.

  The second point is a very important difference between the two. Unlike CSS, Javascript needs to be added to the page after the HTML and CSS have loaded. If you do not do this, depending on the speed of your browser, the functionality of the site may load before you can actually see the page's components and you may get some unexpected results. CSS on the other hand, does not face this problem because even if it loads in first, it just waits for the HTML to finish loading to apply the styles. There is no consequence in the CSS loading before elements have fully loaded.
- **Frontend vs Backend:**
- The difference between JS and Flask is that Flask is used for the backend. Javascript is only meant to handle client interactions, it is not supposed to logically process data. Javascript is intentionally limited for security reasons.
- Example: Javascript cannot be used if we want to send data back to a server or retrieve data from a database.
- Example: If you were making a game, the Javascript is only there to process that a character wants to move. It does not process whether or not it was legal or not for that character to move
- Example: Javascript cannot read or write files while Flask can.
- **Variables:**
- Variables are defined using one of three keywords, **var**, **let** and **const**, and the = operator:
- E.g.
    1. var x = 420
    2. let x = 420
    3. const x = 420
- **var** and **let** have minor differences in terms of scoping. It is recommended to use let over var. However, in the past it was common to use var.

- **const** defines a constant variable that will never change after being instantiated. You cannot change a const's value after it has been instantiated otherwise Javascript will throw a TypeError.
- Similarly to Python, Javascript is dynamically typed meaning that you do not have to indicate the type of variable when you create it. Unlike Python, you have to prefix any variable you create with either var, let, or const to indicate that it is a variable.
- **Equality:**
- There are two main ways of checking for equality in Javascript. We can use either the == operator or the === operator. However, the == operator will sometimes produce odd results. **Therefore, it is recommended that you only use the === operator and to never use the == operator.**
- **Comments:**
- Comments are denoted as **//** or **/*...*/**.
- Single line comments start with //.
- Multi-line comments start with /* and end with */.
- **Operators:**
- A table of the different arithmetic operators:

Operator	**Description**
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation
/	Division
%	Modulus (Division Remainder)
++	Increment
--	Decrement

- A table of the different assignment operators:

Operator	**Example**	**Same As**
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y

/=	x /= y	x = x / y
%=	x %= y	x = x % y
**=	x **= y	x = x ** y

- A table of the different comparison operators:

Operator	**Description**
==	equal to
===	equal value and equal type
!=	not equal
!==	not equal value or not equal type
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

- A table of the different logical operators:

Operator	**Description**
&&	logical and
||	logical or
!	logical not

- A table of the different type operators:

Operator	**Description**
typeof	Returns the type of a variable
instanceof	Returns true if an object is an instance of an object type

- **If statements:**
- Syntax:

```
if (condition 1) {
...
}
else if (condition 2){
```

```
…
}
else{
…
}
```

- **Note:** You must have the if block and only 1 if block. You can have as many else if blocks as you want, and you can have at most 1 else block.
- **While Loops:**
- The **while loop** loops through a block of code as long as a specified condition is true.
- Syntax:

```
while (condition) {
  ...
}
```

- E.g.

```
while (i < 10) {
  text += "The number is " + i;
  i++;
}
```

- The do/while loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested.
- Syntax:

```
do{
  …
}
while (condition);
```

- E.g.

```
do {
  text += "The number is " + i;
  i++;
}
while (i < 10);
```

- **For Loops:**
- There are 3 types of for loops:
  1. **for:** Loops through a block of code a number of times.
     - Syntax:

       ```
       for (statement 1; statement 2; statement 3) {
         ...
       }
       ```

     - Statement 1 is executed one time before the execution of the code block.
     - Statement 2 defines the condition for executing the code block.
     - Statement 3 is executed every time after the code block has been executed.
     - E.g.

       ```
       for (i = 0; i < 5; i++) {
         text += "The number is " + i + "<br>";
       }
       ```

2. **for/in:** Loops through the properties of an object.
    - E.g.
    ```
    var person = {fname:"John", lname:"Doe", age:25};
    var text = "";
    var x;
    for (x in person) {
      text += person[x];
    }
    ```
3. **for/of:** Loops through the values of an iterable object. for/of lets you loop over data structures that are iterable such as Arrays, Strings, Maps, NodeLists, and more.
    - Syntax:
    ```
    for (variable of iterable) {
      ...
    }
    ```
    - **Variable:** For every iteration the value of the next property is assigned to the variable. Variable can be declared with const, let, or var.
    - **Iterable:** An object that has iterable properties.
    - E.g.
    ```
    var cars = ['BMW', 'Volvo', 'Mini'];
    var x;

    for (x of cars) {
      document.write(x + "<br >");
    }
    ```

- **Arrays:**
- JavaScript arrays are used to store multiple values in a single variable.
- An **array** is a special variable, which can hold more than one value at a time.
- Javascript arrays do not have to contain elements of the same type.
- Syntax: `var variable_name = [item1, item2, …, itemn];`
- E.g. `var cars = ["Saab", "Volvo", "BMW"];`
- You access an array element by referring to the index number. All arrays start at index 0.
- E.g. `var name = cars[0];`
- **Objects:**
- An object in Javascript is similar to a dictionary in Python. Unlike in Python, the keys of the object must be String but the value can be any valid type. This type of structure is called a **JSON or Javascript Object Notation**.
- E.g.
```
myObject = {
  "firstName": "kevin",
  "lastName": "zhang",
  "age": 69,
  "cool": true
}
```
- The name:values pairs in JavaScript objects are called properties.
- You can access object properties in two ways:
    1. `objectName.propertyName`
    2. `objectName["propertyName"]`

- Objects can also have methods.
- Methods are actions that can be performed on objects.
- Methods are stored in properties as function definitions.
- E.g.
```
var person = {
  firstName: "John",
  lastName : "Doe",
  id       : 5566,
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};
```
- You access an object method with the following syntax:
  objectName.methodName()
- E.g.
  name = person.fullName();
- **Functions:**
- A JavaScript function is a block of code designed to perform a particular task.
- A JavaScript function is executed when something invokes it. There are 3 ways we can invoke a function:
  1. When an event occurs (when a user clicks a button)
  2. When it is invoked (called) from JavaScript code
  3. Automatically (self invoked)
- You declare a function by prefixing it with the function keyword.
- Syntax:
```
function function_name(...){
…
}
```
- E.g.
```
function myFunction(p1, p2) {
  return p1 * p2;   // The function returns the product of p1 and p2
}
```
- Variables declared within a JavaScript function, become local to the function. **Local variables** can only be accessed from within the function. Since local variables are only recognized inside their functions, variables with the same name can be used in different functions. Local variables are created when a function starts, and deleted when the function is completed.
- **Print Statements:**
- JavaScript can output data in different ways:
  - Writing into an HTML element, using **innerHTML**.
  - Writing into the HTML output using **document.write()**.
  - Writing into an alert box, using **window.alert()**.
  - Writing into the browser console, using **console.log()**.
    Note: You can view console output in Chrome by pressing CMD+OPTION+J on a Mac or CTRL+SHIFT+J on Windows.
- Using INNERHTML:
  - To access an HTML element, JavaScript can use the document.getElementById(id) method. This is the most common way of obtaining

an element to modify. Since IDs in HTML are guaranteed to be unique, we can access a pre-existing element in an HTML form using this strategy.

- The id attribute defines the HTML element. The innerHTML property defines the HTML content.
- E.g.

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Web Page</h1>
<p>My First Paragraph</p>

<p id="demo"></p>

<script>
document.getElementById("demo").innerHTML = 5 + 6;
</script>

</body>
</html>
```

- E.g.

HTML (index.html):

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>My Website</title>
</head>
<body>
  <!-- we will not see the text here because it will be overwritten
   by the Javascript -->
  <p id="my-content">This is the original content</p>
  <script src="script.js"></script>
</body>
</html>
```

Javascript (script.js):

```
document.getElementById('my-content').innerHTML = 'keviniscool';
```

- **Note:** There is a similar command called **document.getElementsByClassName()** however since classes are not unique, this command will always return the result in an array.

- Using document.write():
    - For testing purposes, it is convenient to use document.write().
    - **Note:** Using document.write() after an HTML document is loaded, will delete all existing HTML.
    - E.g.

```
<!DOCTYPE html>
<html>
<body>
```

```
<h1>My First Web Page</h1>
<p>My first paragraph.</p>

<button type="button" onclick="document.write(5 + 6)">Try it</button>

</body>
</html>
```

- Using window.alert():
    - You can use an alert box to display data.
    - E.g.

```
<!DOCTYPE html>
<html>
<body>
<h1>My First Web Page</h1>
<p>My first paragraph.</p>
<script>
window.alert(5 + 6);
</script>
</body>
</html>
```

- Using console.log()
    - For debugging purposes, you can use the console.log() method to display data.
    - E.g.

```
<!DOCTYPE html>
<html>
<body>
<script>
console.log(5 + 6);
</script>
</body>
</html>
```

- **Events:**
- An **HTML event** can be something the browser does or something a user does.
- Here are some examples of HTML events:
    - An HTML web page has finished loading
    - An HTML input field was changed
    - An HTML button was clicked
- Often, when events happen, you may want to do something. JavaScript lets you execute code when events are detected.
- HTML allows event handler attributes, along with JavaScript code, to be added to HTML elements.
- A table of HTML event:

Event	**Description**
onchange	An HTML element has been changed
onclick	The user clicks an HTML element

onmouseover	The user moves the mouse over an HTML element
onmouseout	The user moves the mouse away from an HTML element
onkeydown	The user pushes a keyboard key
onload	The browser has finished loading the page

- Event handlers can be used to handle, and verify, user input, user actions, and browser actions:
    - Things that should be done every time a page loads.
    - Things that should be done when the page is closed.
    - Action that should be performed when a user clicks a button.
    - Content that should be verified when a user inputs data.
- Many different methods can be used to let JavaScript work with events:
    - HTML event attributes can execute JavaScript code directly.
    - HTML event attributes can call JavaScript functions.
    - You can assign your own event handler functions to HTML elements.
    - You can prevent events from being sent or being handled.
- E.g. In the past labs, we've made forms that don't do anything. Using Javascript, we can retrieve input from forms and perform interactive behaviour with the input we've collected. The most common way of doing this is by accessing the .value attribute of an element and reacting to an event such as onclick().

  HTML (index.html):
```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>My Website</title>
</head>
<body>
    <label>What is your name?</label>
    <input type="text" id="name-form" placeholder="Enter your name">
    <button type="submit" onclick="nameResult()">Submit</button>
    <!-- we put a placeholder <p> here so that it can be written into
    later -->
    <!-- using document.write() would overwrite the entire body -->
    <p id="result"></p>
    <script src="script.js"></script>
</body>
</html>
```
  Javascript (script.js):
```
// this function is called whenever the button is clicked
function nameResult() {
  // get the value inside the form
  name = document.getElementById('name-form').value;
  // add the value
  document.getElementById('result').innerHTML = 'your name is: ' + name;
}
```