# Coding Cheat Sheet, CSS, Visual Design, Accessibility, Responsive, Web Design Principles, Flexbox, Grid, SASS

CSS or cascading style sheets is the meat that is added to the raw HTML “skeleton” structure.

# CSS Cascading Style Sheets

// place above HTML or in separate file

<style>

h2 (color:red);

.blue-class {

color: blue;

**font-size**: 30px;

**font-family**: Sans-serif, Helvetica; //changes to Helvetica if Sans-serif is not present

}

.image-class{ // remember can put more than one style in element

border-color:red;

border-width:5px;

border-style:solid;

}

#image-id{

background-color:green;

}

</style>

**CSS Selectors**

    Universal selector \*{} everything on page  
    Type selector h1, h2, h3 {} matches all element names  
    class selector .doc{} or p.note{} matches all classes, can combine type and class  
    id selector #id{}  
    child selector li>a{} all <a> elements children of li  
    descendant selector p a{} targets all <a> elements in p, even if other elements in p  
    adjacent sibling selector h1+p{} matches first p element, after h1 element  
    general sibling selector h1~p{} matches element sibling of another but may not be directly preceding element.

How CSS Rules Cascade  
If two or more rules on same element, which one takes precedence.  
  
    Last Rule: if selectors identical, last one takes precedence  
    Specificity: if one selector more specific than others, more specific one will take precedence e.g. h1 over \* or p b{} more than p  
     !important written after any property will override other rules.  
  
Inheritance  
font-family and color properties inherited automatically by child elements, but background-color and border properties not.  
Can force inheritance by using "inherit" value.  
e.g.  
padding: inherit  
  
Better to use CSS on external sheet.  
  
Important to test CSS in other browsers and on different OS, because CSS is subject to variations in browser and OS code.   
CSS bug list: quirksmode.org (old)

**Over-rides**

ids will always override styles

inline styles will override ids

!important in CSS will override all else

Colorcolor: rgb{xx,xx,xx}  
color: #ee3e80 // hex  
color: red // colornames

**Styling**

font-weight: normal or bold.  
  
font-style: normal, italic or oblique. If browser can't find an italic version of font, it will use oblique algorithm.  
  
text-transform: uppercase, or lowercase, or capitalized (first letter of each word becomes capital). If use uppercase then also try to use letter-spacing quality.  
  
text-decoration: none, underline, overline, line-through, blink (will flash on and off but very annoying).  
  
line-height: (leading pronounced ledding), space from the descending part of text to ascending. Good to have around1.4em to 1.5em. Best given in terms of ems so relative to size of letters.  
  
letter-spacing: (should change if using bold or uppercase text for readability)  
  
word-spacing:  (default 0.25em)  
  
text-align: left, right, center, justify (takes full line except last line)  
  
vertical-align: baseline, sub, super, top, text-top, middle, bottom, text-bottom. (used with inline elements such as img, em, strong. Not to actually make text appear high or low on their own)  
  
text-indent: -9999px (used to push text off screen usually, for example if your heading was an image but you still want it to appear for search engines or screen readers. Otherwise can be used to give some space)  
  
text-shadow: left-to-right-shadow-length top-to-bottom-length amount-of-blur color

**Pseudo-Classes**

:link{  
   change property of links that have not been visited yet  
}  
  
:visited{  
   change property of links already visited  
}  
  
:hover{  
   change properties of elements when hovering over it.  
}  
  
:active{  
   change properties of elements when element is being activated such as button when being pressed  
}  
  
:focus{  
   when elements are ready to be used, such as text inputs.  
}

These previous pseudo-classes should be used in this order, :link, :visited, :hover, :focus, :active.

**Font-Sizes**  
The two main types of length units are absolute and relative. Absolute units tie to physical units of length. For example, in and mm refer to inches and millimeters, respectively. Absolute length units approximate the actual measurement on a screen, but there are some differences depending on a screen's resolution.

Relative units, such as em or rem, are relative to another length value. For example, em is based on the size of an element's font. If you use it to set the font-size property itself, it's relative to the parent's font-size.

**CSS Variable**

To create a CSS Variable, you just need to give it a name with two dashes in front of it and assign it a value like this: **--penguin-skin:gray**;

To use CSS Variable:

**attribute:var(--css-variable-name);**

Fallback variable if someone is using older browser, as CSS Variables are only for CSS3:

background:var(--penguin-skin, black);

When you create a variable, it becomes available for you to use inside the element in which you create it. It also becomes available within any elements nested within it. This effect is known as cascading.

:root{

//css variable declarations

}

Media Query

@media (max-width: 350px) {

// stuff inside

}

# Applied Visual Design

Text  
Two groups of properties:

* those that affect font and its appearance
* those that have same effect on text no matter what font

3 types of fonts:  
serif  
have extra details, traditionally used for long passages of text  
sans-serif  
straight ends to letters, used for small print  
monospace  
have fixed sized space, used to display code

**Text-Align**

text-align: justify; causes all lines of text except the last line to meet the left and right edges of the line box.

text-align: center; centers the text

text-align: right; right-aligns the text

And text-align: left; (the default) left-aligns the text.

You can then over-write these variables by setting them again within a specific element.

<strong> used for UI, not just visual </strong>

<u>underline</u>

**Note**  
Try to avoid using the u tag when it could be confused for a link. Anchor tags also have a default underlined formatting.

<del> line-through</del> equivalent to text-decoration: line-through;

<hr> horizontal line underneath containing element.

box-shadow: offset-x, offset-y, blur-radius, spread-radius

**text-transform**

|  |  |
| --- | --- |
| lowercase | "transform me" |
| uppercase | "TRANSFORM ME" |
| capitalize | "Transform Me" |
| initial | Use the default value |
| inherit | Use the text-transform value from the parent element |
| none | **Default:** Use the original text |

font-weight: (pxs) how thick your characters are.

line-height: (pxs) height of each line, not characters.

element:hover{

//rules for when mouse is hovering over element.

}

**Relative Positioning**

CSS treats each HTML element as its own box, which is usually referred to as the CSS Box Model. Block-level items automatically start on a new line (think headings, paragraphs, and divs) while inline items sit within surrounding content (like images or spans). The default layout of elements in this way is called the normal flow of a document, but CSS offers the position property to override it.

When the position of an element is set to relative, it allows you to specify how CSS should move it *relative* to its current position in the normal flow of the page. It pairs with the CSS offset properties of left or right, and top or bottom. These say how many pixels, percentages, or ems to move the item *away* from where it is normally positioned. The following example moves the paragraph 10 pixels away from the bottom:

**Absolute Positioning**

The next option for the CSS position property is absolute, which locks the element in place relative to its parent container. Unlike the relative position, this removes the element from the normal flow of the document, so surrounding items ignore it. The CSS offset properties (top or bottom and left or right) are used to adjust the position.

One nuance with absolute positioning is that it will be locked relative to its closest *positioned* ancestor. If you forget to add a position rule to the parent item, (this is typically done using position: relative;), the browser will keep looking up the chain and ultimately default to the body tag.

**Fixed Positioning**

The next layout scheme that CSS offers is the fixed position, which is a type of absolute positioning that locks an element relative to the browser window. Similar to absolute positioning, it's used with the CSS offset properties and also removes the element from the normal flow of the document. Other items no longer "realize" where it is positioned, which may require some layout adjustments elsewhere.

One key difference from the absolute position is that the element won't move when the user scrolls.

**Float**

The next positioning tool does not actually use position, but sets the float property of an element. Floating elements are removed from the normal flow of a document and pushed to either the left or right of their containing parent element. It's commonly used with the width property to specify how much horizontal space the floated element requires.

**Z-index**

When elements are positioned to overlap, the element coming later in the HTML markup will, by default, appear on the top of the other elements. However, the z-index property can specify the order of how elements are stacked on top of one another. It must be an integer (i.e. a whole number and not a decimal), and higher values for the z-index property of an element move it higher in the stack than those with lower values.

**Centering Using Margins**

Another positioning technique is to center a block element horizontally. One way to do this is to set its margin to a value of auto.

This method works for images, too. Images are inline elements by default, but can be changed to block elements when you set the display property to block.

**Colours**

Color theory and its impact on design is a deep topic and only the basics are covered in the following challenges. On a website, color can draw attention to content, evoke emotions, or create visual harmony. Using different combinations of colors can really change the look of a website, and a lot of thought can go into picking a color palette that works with your content.

The color wheel is a useful tool to visualize how colors relate to each other - it's a circle where similar hues are neighbors and different hues are farther apart. When two colors are opposite each other on the wheel, they are called complementary colors. They have the characteristic that if they are combined, they "cancel" each other out and create a gray color. However, when placed side-by-side, these colors appear more vibrant and produce a strong visual contrast.

Some examples of complementary colors with their hex codes are:

red (#FF0000) and cyan (#00FFFF)  
green (#00FF00) and magenta (#FF00FF)  
blue (#0000FF) and yellow (#FFFF00)

There are many color picking tools available online that have an option to find the complement of a color.

**Note**  
For all color challenges: Using color can be a powerful way to add visual interest to a page. However, color alone should not be used as the only way to convey important information because users with visual impairments may not understand that content. This issue will be covered in more detail in the Applied Accessibility challenges.

opposite colors on the color wheel can make each other appear more vibrant when placed side-by-side. However, the strong visual contrast can be jarring if it's overused on a website, and can sometimes make text harder to read if it's placed on a complementary-colored background. In practice, one of the colors is usually dominant and the complement is used to bring visual attention to certain content on the page.

**Tertiary Colours**

Computer monitors and device screens create different colors by combining amounts of red, green, and blue light. This is known as the RGB additive color model in modern color theory. Red (R), green (G), and blue (B) are called primary colors. Mixing two primary colors creates the secondary colors cyan (G + B), magenta (R + B) and yellow (R + G). You saw these colors in the Complementary Colors challenge. These secondary colors happen to be the complement to the primary color not used in their creation, and are opposite to that primary color on the color wheel. For example, magenta is made with red and blue, and is the complement to green.

Tertiary colors are the result of combining a primary color with one of its secondary color neighbors. For example, red (primary) and yellow (secondary) make orange. This adds six more colors to a simple color wheel for a total of twelve.

There are various methods of selecting different colors that result in a harmonious combination in design. One example that can use tertiary colors is called the split-complementary color scheme. This scheme starts with a base color, then pairs it with the two colors that are adjacent to its complement. The three colors provide strong visual contrast in a design, but are more subtle than using two complementary colors.

Here are three colors created using the split-complement scheme:

| **Color** | **Hex Code** |
| --- | --- |
| orange | #FF7D00 |
| cyan | #00FFFF |
| raspberry | #FF007D |

**HSL (hue, saturation, lightness)**

Colors have several characteristics including hue, saturation, and lightness. CSS3 introduced the hsl() property as an alternative way to pick a color by directly stating these characteristics.

**Hue** is what people generally think of as 'color'. If you picture a spectrum of colors starting with red on the left, moving through green in the middle, and blue on right, the hue is where a color fits along this line. In hsl(), hue uses a color wheel concept instead of the spectrum, where the angle of the color on the circle is given as a value between 0 and 360.

**Saturation** is the amount of gray in a color. A fully saturated color has no gray in it, and a minimally saturated color is almost completely gray. This is given as a percentage with 100% being fully saturated.

**Lightness** is the amount of white or black in a color. A percentage is given ranging from 0% (black) to 100% (white), where 50% is the normal color.

Here are a few examples of using hsl() with fully-saturated, normal lightness colors:

| **Color** | **HSL** |
| --- | --- |
| red | hsl(0, 100%, 50%) |
| yellow | hsl(60, 100%, 50%) |
| green | hsl(120, 100%, 50%) |
| cyan | hsl(180, 100%, 50%) |
| blue | hsl(240, 100%, 50%) |
| magenta | hsl(300, 100%, 50%) |

**Tonality:** The hsl() option in CSS also makes it easy to adjust the tone of a color. Mixing white with a pure hue creates a tint of that color, and adding black will make a shade. Alternatively, a tone is produced by adding gray or by both tinting and shading. Recall that the 's' and 'l' of hsl() stand for saturation and lightness, respectively. The saturation percent changes the amount of gray and the lightness percent determines how much white or black is in the color. This is useful when you have a base hue you like, but need different variations of it.

**Linear Gradient**

Applying a color on HTML elements is not limited to one flat hue. CSS provides the ability to use color transitions, otherwise known as gradients, on elements. This is accessed through the background property's linear-gradient() function. Here is the general syntax:

background: linear-gradient(gradient\_direction, color 1, color 2, color 3, ...);

The first argument specifies the direction from whicTable Styling Tips:

* give cells padding
* distinguish headings by making uppercase, underline, or background color
* shade alternate rows

align numerals to righth color transition starts - it can be stated as a degree, where 90deg makes a vertical gradient and 45deg is angled like a backslash. The following arguments specify the order of colors used in the gradient.

Example:

background: linear-gradient(90deg, red, yellow, rgb(204, 204, 255));

**Repeating Linear Gradient**

The repeating-linear-gradient() function is very similar to linear-gradient() with the major difference that it repeats the specified gradient pattern. repeating-linear-gradient() accepts a variety of values, but for simplicity, you'll work with an angle value and color stop values in this challenge.

The angle value is the direction of the gradient. Color stops are like width values that mark where a transition takes place, and are given with a percentage or a number of pixels.

In the example demonstrated in the code editor, the gradient starts with the color yellow at 0 pixels which blends into the second color blue at 40 pixels away from the start. Since the next color stop is also at 40 pixels, the gradient immediately changes to the third color green, which itself blends into the fourth color value red as that is 80 pixels away from the beginning of the gradient.

For this example, it helps to think about the color stops as pairs where every two colors blend together.

0px [yellow -- blend -- blue] 40px [green -- blend -- red] 80px

If every two color stop values are the same color, the blending isn't noticeable because it's between the same color, followed by a hard transition to the next color, so you end up with stripes.

**Background Image**

One way to add texture and interest to a background and have it stand out more is to add a subtle pattern. The key is balance, as you don't want the background to stand out too much, and take away from the foreground. The background property supports the url() function in order to link to an image of the chosen texture or pattern. The link address is wrapped in quotes inside the parentheses.

**Transform**

transform:scale(X); where X is the number of times you want the element to be scaled to.

transform:skewX(Xdeg); skews the element in X axis.

transform:skewY(Xdeg); skews element in Y axis.

**::before ::after**

inserts content before or after the element using only css.

**Animation**

To animate an element, you need to know about the animation properties and the @keyframes rule. The animation properties control how the animation should behave and the @keyframes rule controls what happens during that animation. There are eight animation properties in total. This challenge will keep it simple and cover the two most important ones first:

animation-name sets the name of the animation, which is later used by @keyframes to tell CSS which rules go with which animations.

animation-duration sets the length of time for the animation.

@keyframes is how to specify exactly what happens within the animation over the duration. This is done by giving CSS properties for specific "frames" during the animation, with percentages ranging from 0% to 100%. If you compare this to a movie, the CSS properties for 0% is how the element displays in the opening scene. The CSS properties for 100% is how the element appears at the end, right before the credits roll. Then CSS applies the magic to transition the element over the given duration to act out the scene. Here's an example to illustrate the usage of @keyframes and the animation properties:

#anim {  
  animation-name: colorful;  
  animation-duration: 3s;

//The animation-fill-mode specifies the style applied to an element when the animation has finished. // set to forward if want change to not repeat.

// if want to **repeat**, animation-iteration-count:X (or infinite);

// to set motion style, using **animation-timing-function** (see below)  
}  
@keyframes colorful {  
  0% {  
    background-color: blue;

// can set **movement** if position is set, and left, top, right, bottom values are set.  
  }  
  100% {  
    background-color: yellow;

// can **fade** elements using opacity option.   
  }  
}

For the element with the anim id, the code snippet above sets the animation-name to colorful and sets the animation-duration to 3 seconds. Then the @keyframes rule links to the animation properties with the name colorful. It sets the color to blue at the beginning of the animation (0%) which will transition to yellow by the end of the animation (100%). You aren't limited to only beginning-end transitions, you can set properties for the element for any percentage between 0% and 100%.

Note that ms stands for milliseconds, where 1000ms is equal to 1s.

**Animation-timing-function**

In CSS animations, the animation-timing-function property controls how quickly an animated element changes over the duration of the animation. If the animation is a car moving from point A to point B in a given time (your animation-duration), the animation-timing-function says how the car accelerates and decelerates over the course of the drive.

There are a number of predefined keywords available for popular options. For example, the default value is ease, which starts slow, speeds up in the middle, and then slows down again in the end. Other options include ease-out, which is quick in the beginning then slows down, ease-in, which is slow in the beginning, then speeds up at the end, or linear, which applies a constant animation speed throughout.

In CSS animations, Bezier curves are used with the cubic-bezier function. The shape of the curve represents how the animation plays out. The curve lives on a 1 by 1 coordinate system. The X-axis of this coordinate system is the duration of the animation (think of it as a time scale), and the Y-axis is the change in the animation.

The cubic-bezier function consists of four main points that sit on this 1 by 1 grid: p0, p1, p2, and p3. p0 and p3 are set for you - they are the beginning and end points which are always located respectively at the origin (0, 0) and (1, 1). You set the x and y values for the other two points, and where you place them in the grid dictates the shape of the curve for the animation to follow. This is done in CSS by declaring the x and y values of the p1 and p2 "anchor" points in the form: (x1, y1, x2, y2). Pulling it all together, here's an example of a Bezier curve in CSS code:

animation-timing-function: cubic-bezier(0.25, 0.25, 0.75, 0.75);

Although the cubic Bezier curve is mapped on an 1 by 1 coordinate system, and it can only accept x values from 0 to 1, the y value can be set to numbers larger than one. This results in a bouncing movement that is ideal for simulating the juggling ball.

**Table Styling Tips**

* give cells padding
* distinguish headings by making uppercase, underline, or background color
* shade alternate rows
* align numerals to right

# Accessibility

"Accessibility" generally means having web content and a user interface that can be understood, navigated, and interacted with by a broad audience. This includes people with visual, auditory, mobility, or cognitive disabilities. Websites should be open and accessible to everyone, regardless of a user's abilities or resources. Some users rely on assistive technology such as a screen reader or voice recognition software. Other users may be able to navigate through a site only using a keyboard. Keeping the needs of various users in mind when developing your project can go a long way towards creating an open web. Here are three general concepts this section will explore throughout the following challenges:

1. have well-organized code that uses appropriate markup
2. ensure text alternatives exist for non-text and visual content
3. create an easily-navigated page that's keyboard-friendly

Having accessible web content is an ongoing challenge. A great resource for your projects going forward is the **W3 Consortium's Web Content Accessibility Guidelines (WCAG)**. They set the international standard for accessibility and provide a number of criteria you can use to check your work.

**Alt on Images**  
Place an alt=”description” on image so that if image doesn’t load, then description can be seen. Also useful for people using screen readers. If image has already been described, and feels unnecessary, leave description blank alt=””.

**Headings**

Make sure that your headings follow h1, h2, h3 … h6 in order. Use CSS to resize and style. One final point, each page should always have **one (and only one) h1 element**, which is the main subject of your content.

**HTML5 Tags**

<header>//For nav bar, and info

<nav>

</nav>

</header>

<main>

<section>

<article>Article1</article>

</section>

</main>

<footer>//copyright info</footer>

**Audio Tag**The audio tag supports the controls attribute. This shows the browser default play, pause, and other controls, and supports keyboard functionality. This is a boolean attribute, meaning it doesn't need a value, its presence on the tag turns the setting on.

Here's an example:

<audio id="meowClip" controls>  
  <source src="audio/meow.mp3" type="audio/mpeg" />  
  <source src="audio/meow.ogg" type="audio/ogg" />  
</audio>

**Figure**

<figure>

<img/>

<figcaption>//Something</figcaption>

</figure>

**For Attribute in Forms**

The label tag wraps the text for a specific form control item, usually the name or label for a choice. This ties meaning to the item and makes the form more readable. The for attribute on a label tag explicitly associates that label with the form control and is used by screen readers.

The value of the for attribute must be the same as the value of the id attribute of the form control. Here's an example:

<form>  
  <label for="name">Name:</label>  
  <input type="text" id="name" name="name">  
</form>

**Fieldset**The fieldset tag surrounds the entire grouping of radio buttons to achieve this. It often uses a legend tag to provide a description for the grouping, which is read by screen readers for each choice in the fieldset element.

The fieldset wrapper and legend tag are not necessary when the choices are self-explanatory, like a gender selection. Using a label with the for attribute for each radio button is sufficient.

Here's an example:

<form>  
  <fieldset>  
    <legend>Choose one of these three items:</legend>  
    <input id="one" type="radio" name="items" value="one">  
    <label for="one">Choice One</label><br>  
    <input id="two" type="radio" name="items" value="two">  
    <label for="two">Choice Two</label><br>  
    <input id="three" type="radio" name="items" value="three">  
    <label for="three">Choice Three</label>  
  </fieldset>  
</form>

**Form Input Date**Forms often include the input field, which can be used to create several different form controls. The type attribute on this element indicates what kind of input will be created.

You may have noticed the text and submit input types in prior challenges, and HTML5 introduced an option to specify a date field. Depending on browser support, a date picker shows up in the input field when it's in focus, which makes filling in a form easier for all users.

For older browsers, the type will default to text, so it helps to show users the expected date format in the label or as placeholder text just in case.

**Form Time Standardisation**Continuing with the date theme, HTML5 also introduced the time element along with a datetime attribute to standardize times. This is an inline element that can wrap a date or time on a page. A valid format of that date is held by the datetime attribute. This is the value accessed by assistive devices. It helps avoid confusion by stating a standardized version of a time, even if it's written in an informal or colloquial manner in the text.

Here's an example:

<p>Master Camper Cat officiated the cage match between Goro and Scorpion <time datetime="2013-02-13">last Wednesday</time>, which ended in a draw.</p>

**Visually Hide Content Made Just For Screen-Readers**However, CSS's magic can also improve accessibility on your page when you want to visually hide content meant only for screen readers. This happens when information is in a visual format (like a chart), but screen reader users need an alternative presentation (like a table) to access the data. CSS is used to position the screen reader-only elements off the visual area of the browser window.

Here's an example of the CSS rules that accomplish this:

.sr-only {  
  position: absolute;  
  left: -10000px;  
  width: 1px;  
  height: 1px;  
  top: auto;  
  overflow: hidden;  
}

**Note**  
The following CSS approaches will NOT do the same thing:

 display: none; or visibility: hidden; hides content for everyone, including screen reader users

 Zero values for pixel sizes, such as width: 0px; height: 0px; removes that element from the flow of your document, meaning screen readers will ignore it

**High Contrast For Visually Impaired**Low contrast between the foreground and background colors can make text difficult to read. Sufficient contrast improves the readability of your content, but what exactly does "sufficient" mean?

The Web Content Accessibility Guidelines (WCAG) recommend at least a 4.5 to 1 contrast ratio for normal text. The ratio is calculated by comparing the relative luminance values of two colors. This ranges from 1:1 for the same color, or no contrast, to 21:1 for white against black, the strongest contrast. There are many contrast checking tools available online that calculate this ratio for you.

**Sufficient Contrast Using HSL Values**Color is a large part of visual design, but its use introduces two accessibility issues. First, **color alone should not be used as the only way to convey important information because screen reader users won't see it**. Second, foreground and background colors need sufficient contrast so colorblind users can distinguish them.

Previous challenges covered having text alternatives to address the first issue. The last challenge introduced contrast checking tools to help with the second. The WCAG-recommended contrast ratio of 4.5:1 applies for color use as well as gray-scale combinations.

Colorblind users have trouble distinguishing some colors from others - usually in hue but sometimes lightness as well. You may recall the contrast ratio is calculated using the relative luminance (or lightness) values of the foreground and background colors.

In practice, the 4.5:1 ratio can be reached by darkening the darker color and lightening the lighter one with the aid of a color contrast checker. Darker colors on the color wheel are considered to be blues, violets, magentas, and reds, whereas lighter colors are oranges, yellows, greens, and blue-greens.

**Picking Colours for Colour Blindness**There are various forms of colorblindness. These can range from a reduced sensitivity to a certain wavelength of light to the inability to see color at all. The most common form is a reduced sensitivity to detect greens.

For example, if two similar green colors are the foreground and background color of your content, a colorblind user may not be able to distinguish them. Close colors can be thought of as neighbors on the color wheel, and those combinations should be avoided when conveying important information.

**Note**  
Some online color picking tools include visual simulations of how colors appear for different types of colorblindness. These are great resources in addition to online contrast checking calculators.

**Descriptive Links for Screen readers**Screen readers do this by reading the link text, or what's between the anchor (a) tags. Having a list of "click here" or "read more" links isn't helpful. Instead, you should use brief but descriptive text within the a tags to provide more meaning for these users.

**Shortcuts Keys for Focussing**HTML offers the accesskey attribute to specify a shortcut key to activate or bring focus to an element. This can make navigation more efficient for keyboard-only users.

HTML5 allows this attribute to be used on any element, but it's particularly useful when it's used with interactive ones. This includes links, buttons, and form controls.

Here's an example:

<button accesskey="b">Important Button</button>

(for firefox, ALT+ SHIFT+ *accesskey*)

**TabIndex**

The HTML tabindex attribute has three distinct functions relating to an element's keyboard focus. When it's on a tag, it indicates that element can be focused on. The value (an integer that's positive, negative, or zero) determines the behavior.

Certain elements, such as links and form controls, automatically receive keyboard focus when a user tabs through a page. It's in the same order as the elements come in the HTML source markup. This same functionality can be given to other elements, such as div, span, and p, by placing a tabindex="0" attribute on them. Here's an example:

<div tabindex="0">I need keyboard focus!</div>

**Note**  
A negative tabindex value (typically -1) indicates that an element is focusable, but is not reachable by the keyboard. This method is generally used to bring focus to content programmatically (like when a div used for a pop-up window is activated), and is beyond the scope of these challenges.

Bonus - using tabindex also enables the CSS pseudo-class :focus to work on the p tag.

The tabindex attribute also specifies the exact tab order of elements. This is achieved when the value of the attribute is set to a positive number of 1 or higher.

Setting a tabindex="1" will bring keyboard focus to that element first. Then it cycles through the sequence of specified tabindex values (2, 3, etc.), before moving to default and tabindex="0" items.

It's important to note that when the tab order is set this way, it overrides the default order (which uses the HTML source). This may confuse users who are expecting to start navigation from the top of the page. This technique may be necessary in some circumstances, but in terms of accessibility, take care before applying it.

Here's an example:

<div tabindex="1">I get keyboard focus, and I get it first!</div>

<div tabindex="2">I get keyboard focus, and I get it second!</div>

# Responsive Web Design

**Media Queries**Media Queries are a new technique introduced in CSS3 that change the presentation of content based on different viewport sizes. The viewport is a user's visible area of a web page, and is different depending on the device used to access the site.

Media Queries consist of a media type, and if that media type matches the type of device the document is displayed on, the styles are applied. You can have as many selectors and styles inside your media query as you want.

Here's an example of a media query that returns the content when the device's width is less than or equal to 100px:

@media (max-width: 100px) { /\* CSS Rules \*/ }

and the following media query returns the content when the device's height is more than or equal to 350px:

@media (min-height: 350px) { /\* CSS Rules \*/ }

Remember, the CSS inside the media query is applied only if the media type matches that of the device being used.

**Responsive Images**img{

max-width:100%;

display:block;

height:auto;

}

**Sharper “Retina style” Images**The simplest way to make your images appear "retina" (and optimize them for retina displays) is to define their width and height values as only half of what the original file is.

Here is an example of an image that is only using half of the original height and width:

<style>  
  img { height: 250px; width: 250px; }  
</style>  
<img src="coolPic500x500" alt="A most excellent picture">

**Responsive Typography**Instead of using em or px to size text, you can use viewport units for responsive typography. Viewport units, like percentages, are relative units, but they are based off different items. Viewport units are relative to the viewport dimensions (width or height) of a device, and percentages are relative to the size of the parent container element.

The four different viewport units are:

* vw: 10vw would be 10% of the viewport's width.
* vh: 3vh would be 3% of the viewport's height.
* vmin: 70vmin would be 70% of the viewport's smaller dimension (height vs. width).
* vmax: 100vmax would be 100% of the viewport's bigger dimension (height vs. width).

# CSS Flexbox

**Display:Flex;**Adding display: flex to an element turns it into a flex container. This makes it possible to align any children of that element into rows or columns. You do this by adding the flex-direction property to the parent item and setting it to row or column. Creating a row will align the children horizontally, and creating a column will align the children vertically.

Other options for flex-direction are row-reverse and column-reverse.

**Note**  
The default value for the flex-direction property is row.

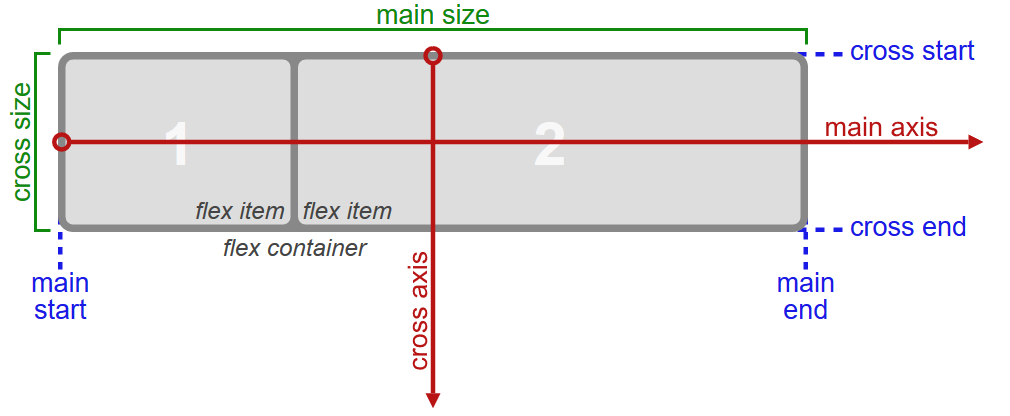
#### Flex-wrap flex-direction:nowrap|wrap|wrap|reverse; to put items onto another line if need be.

**Flex-direction:**

flex-direction: row | row-reverse | column | column-reverse;

**Justify-Content**

Sometimes the flex items within a flex container do not fill all the space in the container. It is common to want to tell CSS how to align and space out the flex items a certain way. Fortunately, the justify-content property has several options to do this. But first, there is some important terminology to understand before reviewing those options.



Recall that setting a flex container as a row places the flex items side-by-side from left-to-right. A flex container set as a column places the flex items in a vertical stack from top-to-bottom. For each, the direction the flex items are arranged is called the **main axis**. For a row, this is a horizontal line that cuts through each item. And for a column, the main axis is a vertical line through the items.

There are several options for how to space the flex items along the line that is the main axis. One of the most commonly used is justify-content: center;, which aligns all the flex items to the center inside the flex container. Others options include:

* flex-start: aligns items to the start of the flex container. For a row, this pushes the items to the left of the container. For a column, this pushes the items to the top of the container.
* flex-end: aligns items to the end of the flex container. For a row, this pushes the items to the right of the container. For a column, this pushes the items to the bottom of the container.
* space-between: aligns items to the center of the main axis, with extra space placed between the items. The first and last items are pushed to the very edge of the flex container. For example, in a row the first item is against the left side of the container, the last item is against the right side of the container, then the other items between them are spaced evenly.
* space-around: similar to space-between but the first and last items are not locked to the edges of the container, the space is distributed around all the items

**Align-items**The align-items property is similar to justify-content. Recall that the justify-content property aligned flex items along the main axis. For rows, the main axis is a horizontal line and for columns it is a vertical line.

Flex containers also have a **cross axis** which is the opposite of the main axis. For rows, the cross axis is vertical and for columns, the cross axis is horizontal.

CSS offers the align-items property to align flex items along the cross axis. For a row, it tells CSS how to push the items in the entire row up or down within the container. And for a column, how to push all the items left or right within the container.

The different values available for align-items include:

* flex-start: aligns items to the start of the flex container. For rows, this aligns items to the top of the container. For columns, this aligns items to the left of the container.
* flex-end: aligns items to the end of the flex container. For rows, this aligns items to the bottom of the container. For columns, this aligns items to the right of the container.
* center: align items to the center. For rows, this vertically aligns items (equal space above and below the items). For columns, this horizontally aligns them (equal space to the left and right of the items).
* stretch: stretch the items to fill the flex container. For example, rows items are stretched to fill the flex container top-to-bottom.
* baseline: align items to their baselines. Baseline is a text concept, think of it as the line that the letters sit on.

**Flex-wrap**CSS flexbox has a feature to split a flex item into multiple rows (or columns). By default, a flex container will fit all flex items together. For example, a row will all be on one line.

However, using the flex-wrap property, it tells CSS to wrap items. This means extra items move into a new row or column. The break point of where the wrapping happens depends on the size of the items and the size of the container.

CSS also has options for the direction of the wrap:

* nowrap: this is the default setting, and does not wrap items.
* wrap: wraps items from left-to-right if they are in a row, or top-to-bottom if they are in a column.
* wrap-reverse: wraps items from right-to-left if they are in a row, or bottom-to-top if they are in a column.

**Flex-Shrink**When it's used, it allows an item to shrink if the flex container is too small. Items shrink when the width of the parent container is smaller than the combined widths of all the flex items within it.

The flex-shrink property takes numbers as values. The higher the number, the more it will shrink compared to the other items in the container. For example, if one item has a flex-shrink value of 1 and the other has a flex-shrink value of 3, the one with the value of 3 will shrink three times as much as the other.

**Flex-Grow**The opposite of flex-shrink is the flex-grow property. Recall that flex-shrink controls the size of the items when the container shrinks. The flex-grow property controls the size of items when the parent container expands.

Using a similar example from the last challenge, if one item has a flex-grow value of 1 and the other has a flex-grow value of 3, the one with the value of 3 will grow three times as much as the other.

**Flex-Basis**The flex-basis property specifies the initial size of the item before CSS makes adjustments with flex-shrink or flex-grow.

The units used by the flex-basis property are the same as other size properties (px, em, %, etc.). The value auto sizes items based on the content.

**Flex Shorthand**There is a shortcut available to set several flex properties at once. The flex-grow, flex-shrink, and flex-basis properties can all be set together by using the flex property.

For example, flex: 1 0 10px; will set the item to flex-grow: 1;, flex-shrink: 0;, and flex-basis: 10px;.

The default property settings are flex: 0 1 auto;.

**Ordering with Flexbox**The order property is used to tell CSS the order of how flex items appear in the flex container. By default, items will appear in the same order they come in the source HTML. The property takes numbers as values, and negative numbers can be used.

**Align-self**The final property for flex items is align-self. This property allows you to adjust each item's alignment individually, instead of setting them all at once. This is useful since other common adjustment techniques using the CSS properties float, clear, and vertical-align do not work on flex items.

align-self accepts the same values as align-items and will override any value set by the align-items property.

# CSS Grid

Turn any HTML element into a grid container by setting its display property to grid. This gives you the ability to use all the other properties associated with CSS Grid.

**Note**  
In CSS Grid, the parent element is referred to as the *container* and its children are called *items*.

**Grid-Template-columns + rows**

Simply creating a grid element doesn't get you very far. You need to define the structure of the grid as well. To add some columns to the grid, use the grid-template-columns property on a grid container as demonstrated below:

.container {  
  display: grid;  
  grid-template-columns: 50px 50px;  
}

This will give your grid two columns that are 50px wide each.

The number of parameters given to the grid-template-columns property indicates the number of columns in the grid, and the value of each parameter indicates the width of each column.

To adjust the rows manually, use the grid-template-rows property in the same way you used grid-template-columns in previous challenge.

**Size of Rows**

You can use absolute and relative units like px and em in CSS Grid to define the size of rows and columns. You can use these as well:

fr: sets the column or row to a fraction of the available space,

auto: sets the column or row to the width or height of its content automatically,

%: adjusts the column or row to the percent width of its container.

Here's the code that generates the output in the preview:

grid-template-columns: auto 50px 10% 2fr 1fr;

This snippet creates five columns. The first column is as wide as its content, the second column is 50px, the third column is 10% of its container, and for the last two columns; the remaining space is divided into three sections, two are allocated for the fourth column, and one for the fifth.

**Grid-Column-Gap + Row**So far in the grids you have created, the columns have all been tight up against each other. Sometimes you want a gap in between the columns. To add a gap between the columns, use the grid-column-gap property like this:

grid-column-gap: 10px;

This creates 10px of empty space between all of our columns.

You can add a gap in between the rows of a grid using grid-row-gap in the same way that you added a gap in between columns in the previous challenge.

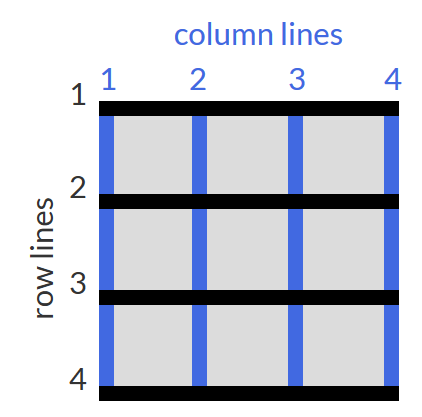
**Grid-Gap**grid-gap is a shorthand property for grid-row-gap and grid-column-gap from the previous two challenges that's more convenient to use. If grid-gap has one value, it will a create a gap between all rows and columns. However, if there are two values, it will use the first one to set the gap between the rows and the second value for the columns.

**Controlling Spacing of Grid Items**

Up to this point, all the properties that have been discussed are for grid containers. The grid-column property is the first one for use on the grid items themselves.

The hypothetical horizontal and vertical lines that create the grid are referred to as *lines*. These lines are numbered starting with 1 at the top left corner of the grid and move right for columns and down for rows, counting upward.

This is what the lines look like for a 3x3 grid:



To control the amount of columns an item will consume, you can use the grid-column property in conjunction with the line numbers you want the item to start and stop at.

Here's an example:

grid-column: 1 / 3;

This will make the item start at the first vertical line of the grid on the left and span to the 3rd line of the grid, consuming two columns.

You define the horizontal lines you want an item to start and stop at using the grid-row property on a grid item.

**Aligning Item in Cell**In CSS Grid, the content of each item is located in a box which is referred to as a *cell*. You can align the content's position within its cell horizontally using the justify-self property on a grid item. By default, this property has a value of stretch, which will make the content fill the whole width of the cell. This CSS Grid property accepts other values as well:

start: aligns the content at the left of the cell,

center: aligns the content in the center of the cell,

end: aligns the content at the right of the cell.

Just as you can align an item horizontally, there's a way to align an item vertically as well. To do this, you use the align-self property on an item. This property accepts all of the same values as justify-self from the last challenge.

**Aligning All Items in Grid**Sometimes you want all the items in your CSS Grid to share the same alignment. You can use the previously learned properties and align them individually, or you can align them all at once horizontally by using justify-items on your grid container. This property can accept all the same values you learned about in the previous two challenges, the difference being that it will move **all** the items in our grid to the desired alignment.

Using the align-items property on a grid container will set the vertical alignment for all the items in our grid.

**Grouping Grid Items in Template**You can group cells of your grid together into an *area* and give the area a custom name. Do this by using grid-template-areas on the container like this:

grid-template-areas:  
  "header header header"  
  "advert content content"  
  "footer footer footer";

The code above merges the top three cells together into an area named header, the bottom three cells into a footer area, and it makes two areas in the middle row; advert and content.

**Note**  
Every word in the code represents a cell and every pair of quotation marks represent a row.

In addition to custom labels, you can use a period (.) to designate an empty cell in the grid.

**Assigning Items to Grid Area**After creating an areas template for your grid container, as shown in the previous challenge, you can place an item in your custom area by referencing the name you gave it. To do this, you use the grid-area property on an item like this:

.item1 { grid-area: header; }

This lets the grid know that you want the item1 class to go in the area named header. In this case, the item will use the entire top row because that whole row is named as the header area.

**Creating Grid-Area without Template**The grid-area property you learned in the last challenge can be used in another way. If your grid doesn't have an areas template to reference, you can create an area on the fly for an item to be placed like this:

item1 { grid-area: 1/1/2/4; }

This is using the line numbers you learned about earlier to define where the area for this item will be. The numbers in the example above represent these values:

grid-area: horizontal line to start at / vertical line to start at / horizontal line to end at / vertical line to end at;

So the item in the example will consume the rows between lines 1 and 2, and the columns between lines 1 and 4.

**Repeat Rows or Columns**When you used grid-template-columns and grid-template-rows to define the structure of a grid, you entered a value for each row or column you created.

Lets say you want a grid with 100 rows of the same height. It isn't very practical to insert 100 values individually. Fortunately, there's a better way - by using the repeat function to specify the number of times you want your column or row to be repeated, followed by a comma and the value you want to repeat.

Here's an example that would create the 100 row grid, each row at 50px tall.

grid-template-rows: repeat(100, 50px);

You can also repeat multiple values with the repeat function, and insert the function amongst other values when defining a grid structure. Here's what I mean:

grid-template-columns: repeat(2, 1fr 50px) 20px;

This translates to:

grid-template-columns: 1fr 50px 1fr 50px 20px;

**Note**  
1fr 50px is repeated twice followed by 20px.

**Minmax function Setting limits on Items**There's another built-in function to use with grid-template-columns and grid-template-rows called minmax. It's used to limit the size of items when the grid container changes size. To do this you need to specify the acceptable size range for your item. Here is an example:

grid-template-columns: 100px minmax(50px, 200px);

In the code above, grid-template-columns is set to create two columns; the first is 100px wide, and the second has the minimum width of 50px and the maximum width of 200px.

**Autofill**The repeat function comes with a option called *auto-fill*. This allows you to automatically insert as many rows or columns of your desired size as possible depending on the size of the container. You can create flexible layouts when combining auto-fill with minmax.

In the preview, grid-template-columns is set to

repeat(auto-fill, minmax(60px, 1fr));

When the container changes size, this setup keeps inserting 60px columns and stretching them until it can insert another one.

**Note**  
If your container can't fit all your items on one row, it will move them down to a new one.

**Autofit**auto-fit works almost identically to auto-fill. The only difference is that when the container's size exceeds the size of all the items combined, auto-fill keeps inserting empty rows or columns and pushes your items to the side, while auto-fit collapses those empty rows or columns and stretches your items to fit the size of the container.

**Note**  
If your container can't fit all your items on one row, it will move them down to a new one.

**Grid within Grid**Turning an element into a grid only affects the behavior of its direct descendants. So by turning a direct descendant into a grid, you have a grid within a grid.

For example, by setting the display and grid-template-columns properties of the element with the item3 class, you create a grid within your grid.

**SASS**

In Sass, variables start with a $ followed by the variable name.

Here are a couple examples:

$main-fonts: Arial, sans-serif;  
$headings-color: green;  
  
//To use variables:  
h1 {  
  font-family: $main-fonts;  
  color: $headings-color;  
}

One example where variables are useful is when a number of elements need to be the same color. If that color is changed, the only place to edit the code is the variable value.

**Nesting**

Sass allows nesting of CSS rules, which is a useful way of organizing a style sheet.

Normally, each element is targeted on a different line to style it, like so:

nav {  
  background-color: red;  
}  
  
nav ul {  
  list-style: none;  
}  
  
nav ul li {  
  display: inline-block;  
}

For a large project, the CSS file will have many lines and rules. This is where nesting can help organize your code by placing child style rules within the respective parent elements:

nav {  
  background-color: red;  
  
  ul {  
    list-style: none;  
  
    li {  
      display: inline-block;  
    }  
  }  
}

**Mixins**

Mixins are like functions for CSS. Used for code reuse especially when new CSS features take time to be updated into each browser, so you don’t want to write up all of the webkit, moz, and ms properties over and over again. Here is how to write one:

@mixin box-shadow($x, $y, $blur, $c){   
  -webkit-box-shadow: $x, $y, $blur, $c;  
  -moz-box-shadow: $x, $y, $blur, $c;  
  -ms-box-shadow: $x, $y, $blur, $c;  
  box-shadow: $x, $y, $blur, $c;  
}

The definition starts with @mixin followed by a custom name. The parameters (the $x, $y, $blur, and $c in the example above) are optional.

Now any time a box-shadow rule is needed, only a single line calling the mixin replaces having to type all the vendor prefixes. A mixin is called with the @include directive:

div {  
  @include box-shadow(0px, 0px, 4px, #fff);  
}

**@if @else if @else directive**

**The @if directive in Sass is useful to test for a specific case - it works just like the if statement in JavaScript.**

@mixin make-bold($bool) {  
  @if $bool == true {  
    font-weight: bold;  
  }  
}

And just like in JavaScript, @else if and @else test for more conditions:

@mixin text-effect($val) {  
  @if $val == danger {  
    color: red;  
  }  
  @else if $val == alert {  
    color: yellow;  
  }  
  @else if $val == success {  
    color: green;  
  }  
  @else {  
    color: black;  
  }

**@for directive**

The @for directive adds styles in a loop, very similar to a for loop in JavaScript.

@for is used in two ways: "start through end" or "start to end". The main difference is that "start to end" excludes the end number, and "start through end" includes the end number.

Here's a start **through** end example:

@for $i from 1 through 12 {  
  .col-#{$i} { width: 100%/12 \* $i; }  
}

The #{$i} part is the syntax to combine a variable (i) with text to make a string. When the Sass file is converted to CSS, it looks like this:

.col-1 {  
  width: 8.33333%;  
}  
  
.col-2 {  
  width: 16.66667%;  
}  
  
...  
  
.col-12 {  
  width: 100%;  
}

This is a powerful way to create a grid layout. Now you have twelve options for column widths available as CSS classes.

**@each Directive for lists and mapping**

Sass also offers the @each directive which loops over each item in a list or map.

On each iteration, the variable gets assigned to the current value from the list or map.

@each $color in blue, red, green {  
  .#{$color}-text {color: $color;}  
}

A map has slightly different syntax. Here's an example:

$colors: (color1: blue, color2: red, color3: green);  
  
@each $key, $color in $colors {  
  .#{$color}-text {color: $color;}  
}

Note that the $key variable is needed to reference the keys in the map. Otherwise, the compiled CSS would have color1, color2... in it.

Both of the above code examples are converted into the following CSS:

.blue-text {  
  color: blue;  
}  
  
.red-text {  
  color: red;  
}  
  
.green-text {  
  color: green;  
}