mdn web docs_

CSS values and units

Every property used in CSS has a value type defining the set of values that are allowed for that property. Taking a look at any property page on MDN will help you understand the values associated with a value type that are valid for any particular property. In this lesson we will take a look at some of the most frequently used value types, and their most common values and units.

Prerequisites:	Basic computer literacy, basic software installed, basic knowledge of working with files, HTML basics (study Introduction to HTML), and an idea of how CSS works (study CSS first steps).
Objective:	To learn about the different types of values and units used in CSS properties.

What is a CSS value?

In CSS specifications and on the property pages here on MDN you will be able to spot value types as they will be surrounded by angle brackets, such as `<color>` or `<length>`. When you see the value type `<color>` as valid for a particular property, that means you can use any valid color as a value for that property, as listed on the `<color>` reference page.

Note: You'll also see CSS value types referred to as data types. The terms are basically interchangeable — when you see something in CSS referred to as a data type, it is really just a fancy way of saying value type. The term value refers to any particular expression supported by a value type that you choose to use.

Note: Yes, CSS value types tend to be denoted using angle brackets to differentiate them from CSS properties (e.g., the `color` property, versus the `<color>` data type). You might get confused between CSS data types and HTML elements too, as they both use angle brackets, but this is unlikely — they are used in very different contexts.

In the following example, we have set the color of our heading using a keyword, and the background using the `rgb()` function:

```css
h1 {
  color: black;
  background-color: rgb(197, 93, 161);
}
```

A value type in CSS is a way to define a collection of allowable values. This means that if you see `<color>` as valid you don't need to wonder which of the different types of color value can be used — keywords, hex values, `rgb()` functions, etc. You can use any available `<color>` values, assuming they are supported by your browser. The page on MDN for each value will give you information about browser support. For example, if you look at the page for `<color>` you will see that the browser compatibility section lists different types of color values and support for them.

Let's have a look at some of the types of values and units you may frequently encounter, with examples so that you can try out different possible values.

Numbers, lengths, and percentages

There are various numeric value types that you might find yourself using in CSS. The following are all classed as numeric:

Data type	Description
`<integer>`	An `<integer>` is a whole number such as `1024` or `-55`.
`<number>`	A `<number>` represents a decimal number — it may or may not have a

Data type	Description
	decimal point with a fractional component. For example, `0.255`, `128`, or `-1.2`.
`<dimension>`	A `<dimension>` is a `<number>` with a unit attached to it. For example, `45deg`, `5s`, or `10px`. `<dimension>` is an umbrella category that includes the `<length>`, `<angle>`, `<time>`, and `<resolution>` types.
`<percentage>`	A `<percentage>` represents a fraction of some other value. For example, `50%`. Percentage values are always relative to another quantity. For example, an element's length is relative to its parent element's length.

Lengths

The numeric type you will come across most frequently is `<length>`. For example, `10px` (pixels) or `30em`. There are two types of lengths used in CSS — relative and absolute. It's important to know the difference in order to understand how big things will become.

Absolute length units

The following are all absolute length units — they are not relative to anything else, and are generally considered to always be the same size.

Unit	Name	Equivalent to
`cm`	Centimeters	1cm = 37.8px = 25.2/64in
`mm`	Millimeters	1mm = 1/10th of 1cm
`Q`	Quarter-millimeters	1Q = 1/40th of 1cm
`in`	Inches	1in = 2.54cm = 96px
`pc`	Picas	1pc = 1/6th of 1in
`pt`	Points	1pt = 1/72nd of 1in

Unit	Name	Equivalent to
`px`	Pixels	1px = 1/96th of 1in

Most of these units are more useful when used for print, rather than screen output. For example, we don't typically use `cm` (centimeters) on screen. The only value that you will commonly use is `px` (pixels).

Relative length units

Relative length units are relative to something else, perhaps the size of the parent element's font, or the size of the viewport. The benefit of using relative units is that with some careful planning you can make it so the size of text or other elements scales relative to everything else on the page. Some of the most useful units for web development are listed in the table below.

Unit	Relative to
`em`	Font size of the parent, in the case of typographical properties like `font-size`, and font size of the element itself, in the case of other properties like `width`.
`ex`	x-height of the element's font.
`ch`	The advance measure (width) of the glyph "0" of the element's font.
`rem`	Font size of the root element.
`lh`	Line height of the element.
`rlh`	Line height of the root element. When used on the `font-size` or `line-height` properties of the root element, it refers to the properties' initial value.
`vw`	1% of the viewport's width.
`vh`	1% of the viewport's height.
`vmin`	1% of the viewport's smaller dimension.

Unit	Relative to
`vmax`	1% of the viewport's larger dimension.
`vb`	1% of the size of the initial containing block in the direction of the root element's block axis.
`vi`	1% of the size of the initial containing block in the direction of the root element's inline axis.
`svw`, `svh`	1% of the small viewport's width and height, respectively.
`lvw`, `lvh`	1% of the large viewport's width and height, respectively.
`dvw`, `dvh`	1% of the dynamic viewport's width and height, respectively.

Exploring an example

In the example below, you can see how some relative and absolute length units behave. The first box has a `width` set in pixels. As an absolute unit, this width will remain the same no matter what else changes.

The second box has a width set in `vw` (viewport width) units. This value is relative to the viewport width, and so 10vw is 10 percent of the width of the viewport. If you change the width of your browser window, the size of the box should change. However this example is embedded into the page using an `<iframe>`, so this won't work. To see this in action you'll have to try the example after opening it in its own browser tab .

The third box uses `em` units. These are relative to the font size. I've set a font size of `1em` on the containing `<div>`, which has a class of `.wrapper`. Change this value to `1.5em` and you will see that the font size of all the elements increases, but only the last item will get wider, as its width is relative to that font size.

After following the instructions above, try playing with the values in other ways, to see what you get.

ems and rems

`em` and `rem` are the two relative lengths you are likely to encounter most frequently when sizing anything from boxes to text. It's worth understanding how these work, and the differences between them, especially when you start getting on to more complex subjects like styling text or CSS layout. The below example provides a demonstration.

The HTML illustrated below is a set of nested lists — we have three lists in total and both examples have the same HTML. The only difference is that the first has a class of ems and the second a class of rems.

To start with, we set 16px as the font size on the `<html>` element.

To recap, the em unit means "my parent element's font-size" in the case of typography. The `<li>` elements inside the `<ul>` with a `class` of `ems` take their sizing from their parent. So each successive level of nesting gets progressively larger, as each has its font size set to `1.3em` — 1.3 times its parent's font size.

To recap, the rem unit means "The root element's font-size" (rem stands for "root em"). The `<li>` elements inside the `<ul>` with a `class` of `rems` take their sizing from the root element (`<html>`). This means that each successive level of nesting does not keep getting larger.

However, if you change the `<html>` element's `font-size` in the CSS you will see that everything else changes relative to it — both `rem` - and `em` -sized text.

Percentages

In a lot of cases, a percentage is treated in the same way as a length. The thing with percentages is that they are always set relative to some other value. For example, if you set an element's `font-size` as a percentage, it will be a percentage of the `font-size` of the element's parent. If you use a percentage for a `width` value, it will be a percentage of the `width` of the parent.

In the below example the two percentage-sized boxes and the two pixel-sized boxes have the same class names. The sets are 40% and 200px wide respectively.

The difference is that the second set of two boxes is inside a wrapper that is 400 pixels wide. The second 200px wide box is the same width as the first one, but the second 40% box is now 40% of 400px — a lot narrower than the first one!

Try changing the width of the wrapper or the percentage value to see how this works.

The next example has font sizes set in percentages. Each `<li>` has a `font-size` of 80%; therefore, the nested list items become progressively smaller as they inherit their sizing from their parent.

Note that, while many value types accept a length or a percentage, there are some that only accept length. You can see which values are accepted on the MDN property reference pages. If the allowed value includes `<length-percentage>` then you can use a length or a percentage. If the allowed value only includes `<length>`, it is not possible to use a percentage.

Numbers

Some value types accept numbers, without any unit added to them. An example of a property which accepts a unitless number is the `opacity` property, which controls the opacity of an element (how transparent it is). This property accepts a number between `0` (fully transparent) and `1` (fully opaque).

In the below example, try changing the value of **`opacity`** to various decimal values between **`0`** and **`1`** and see how the box and its contents become more or less opaque.

Note: When you use a number in CSS as a value it should not be surrounded in quotes.

Color

There are many ways to specify color in CSS, some of which are more recently implemented than others. The same color values can be used everywhere in CSS, whether you are specifying text color, background color, or whatever else.

The standard color system available in modern computers supports 24-bit colors, which allows the display of about 16.7 million distinct colors via a combination of different red, green and blue channels with 256 different values per channel (256 × 256 × 256 = 16,777,216). Let's have a look at some of the ways in which we can specify colors in CSS.

> Note: In this tutorial we will look at the common methods of specifying color that have good browser support; there are others but they don't have as good support and are less common.

Color keywords

Quite often in examples here in the learn section or elsewhere on MDN you will see the color keywords used, as they are a simple and understandable way of specifying color. There are a number of these keywords, some of which have fairly entertaining names! You can see a full list on the page for the `<color>` value type.

Try playing with different color values in the live examples below, to get more of an idea how they work.

Hexadecimal RGB values

The next type of color value you are likely to encounter is hexadecimal codes. Each hex value consists of a hash/pound symbol (#) followed by six hexadecimal numbers, each of which can take one of 16 values between 0 and f (which represents 15) — so `0123456789abcdef`. Each pair of values represents one of the channels — red, green and blue — and allows us to specify any of the 256 available values for each (16 × 16 = 256).

These values are a bit more complex and less easy to understand, but they are a lot more versatile than keywords — you can use hex values to represent any color you want to use in your color scheme.

Again, try changing the values to see how the colors vary.

RGB and RGBA values

The third scheme we'll talk about here is RGB. An RGB value is a function — `rgb()` — which is given three parameters that represent the red, green, and blue channel values of the colors, in much the same way as hex values. The difference with RGB is that each channel is represented not by two hex digits, but by a decimal number between 0 and 255 — somewhat easier to understand.

Let's rewrite our last example to use RGB colors:

You can pass a fourth parameter to `rgb()`, which represents the alpha channel of the color, which controls opacity. If you set this value to `0` it will make the color fully transparent, whereas `1` will make it fully opaque. Values in between give you different levels of transparency.

> Note: Setting an alpha channel on a color has one key difference to using the `opacity` property we looked at earlier. When you use opacity you make the element and everything inside it opaque, whereas using RGB with an alpha parameter colors only makes the color you are specifying opaque.

In the example below, we have added a background image to the containing block of our colored boxes. We have then set the boxes to have different opacity values — notice how the background shows through more when the alpha channel value is smaller.

In this example, try changing the alpha channel values to see how it affects the color output.

> Note: In older versions of CSS, the `rgb()` syntax didn't support an alpha parameter - you needed to use a different function called `rgba()` for that. These days you can pass an alpha parameter to `rgb()`, but for backwards compatibility with old websites, the `rgba()` syntax is still supported, and has exactly the same behavior as `rgb()`.

HSL and HSLA values

An alternative way to specify colors is the HSL color model. Instead of red, green, and blue values, the `hsl()` function accepts hue, saturation, and lightness values, which are used to distinguish between the 16.7 million colors, but in a different way:

- Hue: The base shade of the color. This takes a value between 0 and 360, representing the angles around a color wheel.
- Saturation: How saturated is the color? This takes a value from 0–100%, where 0 is no color (it will appear as a shade of grey), and 100% is full color saturation
- Lightness: How light or bright is the color? This takes a value from 0–100%, where 0 is no light (it will appear completely black) and 100% is full light (it will appear completely white)

We can update the RGB example to use HSL colors like this:

Just like with `rgb()` you can pass an alpha parameter to `hsl()` to specify opacity:

> Note: In older versions of CSS, the `hsl()` syntax didn't support an alpha parameter - you needed to use a different function called `hsla()` for that. These days you can pass an alpha parameter to `hsl()`, but for backwards compatibility with old websites, the `hsla()` syntax is still supported, and has exactly the same behavior as `hsl()`.

You can use any of these color values in your projects. It is likely that for most projects you will decide on a color palette and then use those colors — and your chosen method of specifying color — throughout the whole project. You can mix and match color models, however for consistency it is usually best if your entire project uses the same one!

Images

The `<image>` value type is used wherever an image is a valid value. This can be an actual image file pointed to via a `url()` function, or a gradient.

In the example below, we have demonstrated an image and a gradient in use as a value for the CSS `background-image` property.

> Note: There are some other possible values for `<image>`, however these are newer and currently have poor browser support. Check out the page on MDN for the `<image>` data type if you want to read about them.

Position

The `<position>` value type represents a set of 2D coordinates, used to position an item such as a background image (via `background-position`). It can take keywords such as `top`, `left`, `bottom`, `right`, and `center` to align items with specific bounds of a 2D box, along with lengths, which represent offsets from the top and left-hand edges of the box.

A typical position value consists of two values — the first sets the position horizontally, the second vertically. If you only specify values for one axis the other will default to `center`.

In the following example we have positioned a background image 40px from the top and to the right of the container using a keyword.

Play around with these values to see how you can push the image around.

Strings and identifiers

Throughout the examples above, we've seen places where keywords are used as a value (for example `<color>` keywords like `red`, `black`, `rebeccapurple`, and `goldenrod`). These keywords are more accurately described as identifiers, a special value that CSS understands. As such they are not quoted — they are not treated as strings.

There are places where you use strings in CSS. For example, when specifying generated content. In this case, the value is quoted to demonstrate that it is a string. In the example below, we use unquoted color keywords along with a quoted generated content string.

Functions

The final type of value we will take a look at is the group of values known as functions. In programming, a function is a reusable section of code that can be run multiple times to complete a repetitive task with minimum effort on the part of both the developer and the computer. Functions are usually associated with languages like JavaScript, Python, or C++, but they do exist in CSS too, as property values. We've already seen functions in action in the Colors section — `rgb()`, `hsl()`, etc. The value used to return an image from a file — `url()` — is also a function.

A value that behaves more like something you might find in a traditional programming language is the `calc()` CSS function. This function gives you the ability to do simple calculations inside your CSS. It's particularly useful if you want to work out values that you can't define when writing the CSS for your project, and need the browser to work out for you at runtime.

For example, below we are using `calc()` to make the box `20% + 100px` wide. The 20% is calculated from the width of the parent container `.wrapper` and so will change if that width changes. We can't do this calculation beforehand because we don't know what 20% of the parent will be, so we use `calc()` to tell the browser to do it for us.

Test your skills!

You've reached the end of this article, but can you remember the most important information? You can find some further tests to verify that you've retained this information before you move on — see Test your skills: Values and units.

Summary

This has been a quick run-through of the most common types of values and units you might encounter. You can have a look at all of the different types on the CSS Values and units reference page — you will encounter many of these in use as you work through these lessons.

The key thing to remember is that each property has a defined list of allowed value types, and each value type has a definition explaining what the values are. You can then look up the specifics here on MDN. For example, understanding that `<image>` also allows you to create a color gradient is useful but perhaps non-obvious knowledge to have!

In the next article, we'll take a look at how items are sized in CSS.