CSS values and units

CSS rules contain declarations, which in turn are composed of properties and values. Each property used in CSS has a **value type** that describes what kind of values it is allowed to have. In this lesson, we will take a look at some of the most frequently used value types, what they are, and how they work.

> **Note:** Each CSS property page has a syntax section that lists the value types you can use with that property.

Prerequisites:	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 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:** CSS value types tend to be enclosed in angle brackets (`<`, `>`) to differentiate them from CSS properties. For example there is a `color` property and a `<color>` data type. This is not to be confused with HTML elements, as they also use angle brackets, but this is something to keep in mind that the context should make clear.

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 decimal point with a fractional component. For example, `0.255`, `128`,

Data type	Description
	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

mdn web docs

(pixels) or `50em`. 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
`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.
`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

Unit	Relative to
	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.

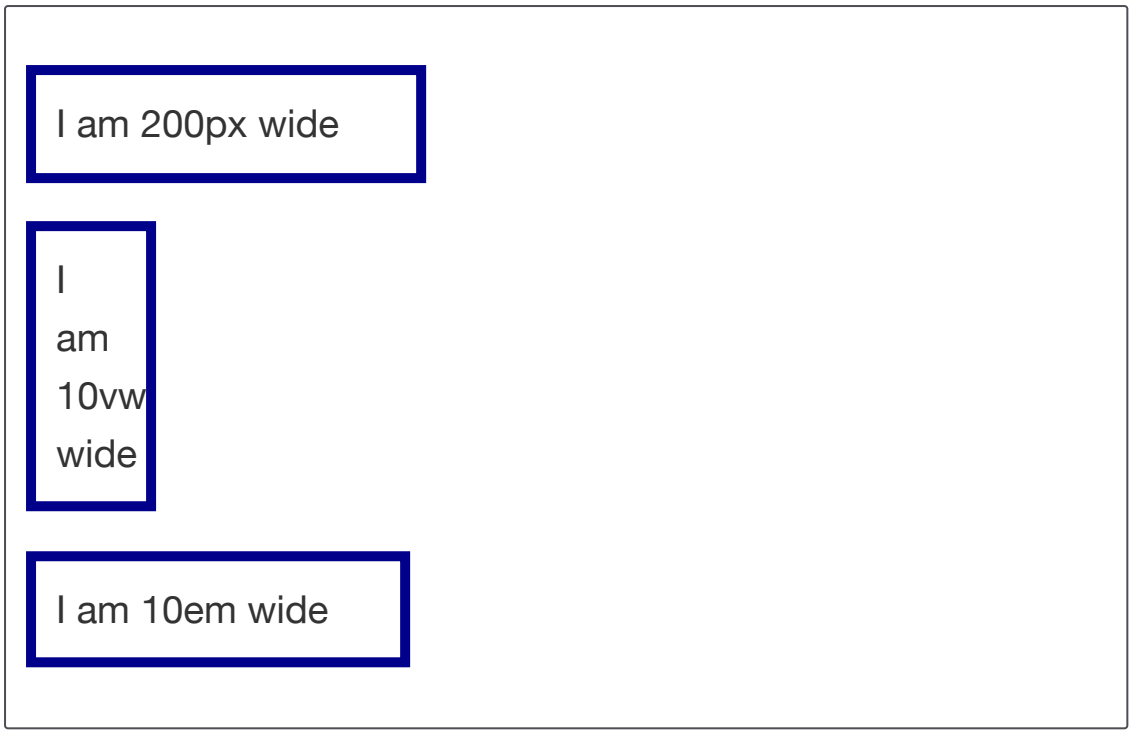
Exploring an example

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.



Interactive editor

```
.wrapper {
  font-size: 1em;
}

.px {
  width: 200px;
}

.vw {
  width: 10vw;
}

.em {
  width: 10em;
}
```

```
<div class="wrapper">
  <div class="box px">I am 200px wide</div>
  <div class="box vw">I am 10vw wide</div>
  <div class="box em">I am 10em wide</div>
</div>
```

Reset

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 two 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.

- One
- Two
- Three
  - Three A
  - Three B
    - Three B 2

- One
- Two
- Three
  - Three A
  - Three B
    - Three B 2

Interactive editor

```
html {
  font-size: 16px;
}

.ems li {
  font-size: 1.3em;
}

.rems li {
  font-size: 1.3rem;
}
```

```
<ul class="ems">
  <li>One</li>
  <li>Two</li>
  <li>Three
    <ul>
      <li>Three A</li>
```

Reset

Line height units

`lh` and `rlh` are relative lengths units similar to `em` and `rem`. The difference between `lh` and `rlh` is that the first one is relative to the line height of the element itself, while the second one is relative to the line height of the root element, usually `<html>`.

Using these units, we can precisely align box decoration to the text. In this example, we use `lh` unit to create notepad-like lines using `repeating-linear-gradient()`. It doesn't matter what's the line height of the text, the lines will always start in the right place.

CSS Play

```
p {
  margin: 0;
  background-image: repeating-linear-gradient(
    to top,
    lightskyblue 0 2px,
    transparent 2px 1lh
  );
}
```

HTML Play

```
<p style="line-height: 2em">
  Summer is a time for adventure, and this year was no exception. I had many
  exciting experiences, but two of my favorites were my trip to the beach and my
  week at summer camp.
</p>

<p style="line-height: 4em">
  At the beach, I spent my days swimming, collecting shells, and building
  sandcastles. I also went on a boat ride and saw dolphins swimming alongside
  us.
</p>
```

Play

Summer is a time for adventure, and this year was no exception. I had many exciting experiences, but two of my favorites were my trip to the beach and my week at summer camp.

At the beach, I spent my days swimming, collecting shells, and building sandcastles. I also went on a boat ride and saw dolphins swimming alongside us.

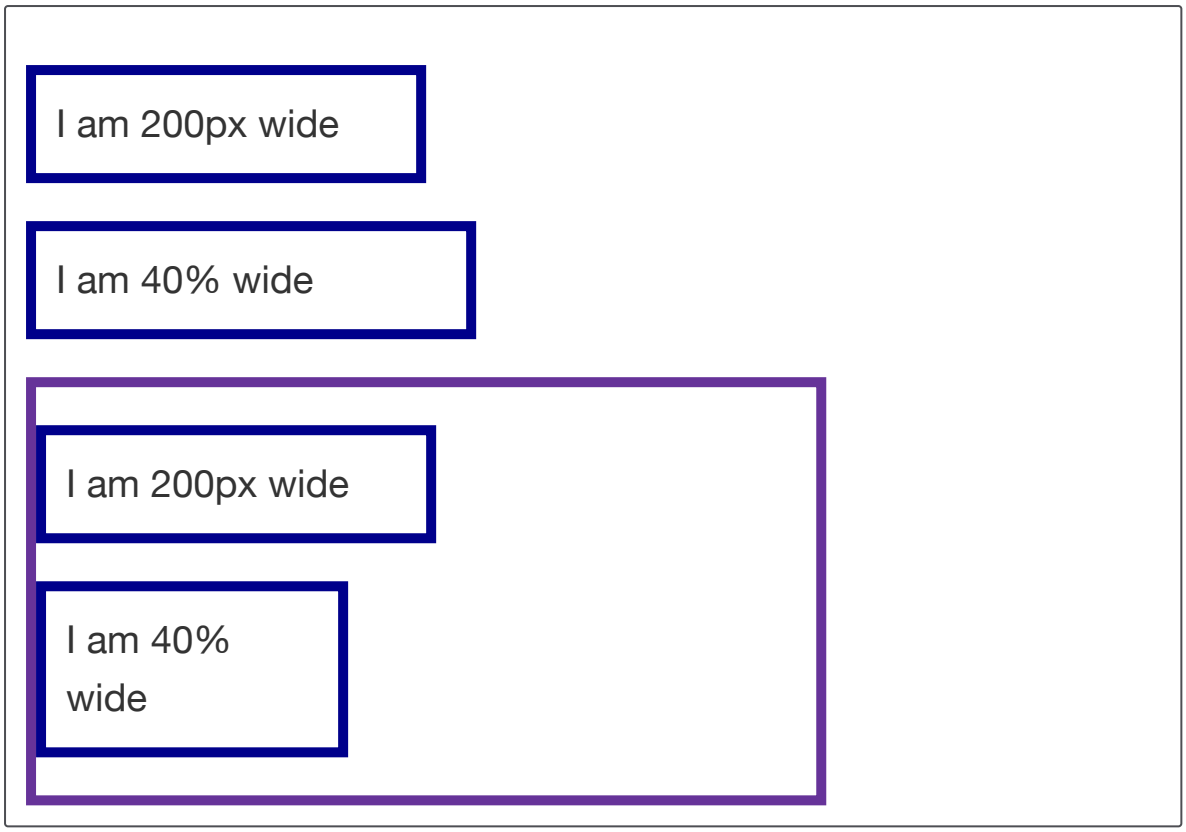
Percentages

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.



Interactive editor

```
.wrapper {
  width: 400px;
  border: 5px solid rebeccapurple;
}

.px {
  width: 200px;
}

.percent {
  width: 40%;
}
```

```
<div class="box px">I am 200px wide</div>
<div class="box percent">I am 40% wide</div>
<div class="wrapper">
  <div class="box px">I am 200px wide</div>
  <div class="box percent">I am 40% wide</div>
</div>
```

Reset

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.

- One
- Two
- Three
  - Three A
  - Three B
    - Three B 2

Interactive editor

```
li {
  font-size: 80%;
}
```

```
<ul>
  <li>One</li>
  <li>Two</li>
  <li>Three
    <ul>
      <li>Three A</li>
      <li>Three B
        <ul>
          <li>Three B 2</li>
        </ul>
      </li>
    </ul>
  </li>
</ul>
```

Reset

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.

Interactive editor

```
.box {
  opacity: 0.6;
}
```

```
<div class="wrapper">
  <div class="box">I am a box with opacity</div>
</div>
```

Reset

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

Color

Color values can be used in many places in CSS, whether you are specifying the color of text, backgrounds, borders, and lots more. There are many ways to set color in CSS, allowing you to control plenty of exciting properties.

The standard color system available in modern computers supports 24-bit colors, which allows displaying about 16.7 million distinct colors via a combination of different red, green, and blue channels with 256 different values per channel (256 x 256 x 256 = 16,777,216).

In this section, we'll first look at the most commonly seen ways of specifying colors: using keywords, hexadecimal, and `rgb()` values. We'll also take a quick look at additional color functions, enabling you to recognize them when you see them or experiment with different ways of applying color.

You will likely decide on a color palette and then use those colors — and your favorite way of specifying color — throughout your project. You can mix and match color models, but it's usually best if your entire project uses the same method of declaring colors for consistency!

Color keywords

You will see the color keywords (or 'named colors') used in many MDN code examples. As the `<named-color>` data type contains a very finite number of color values, these are not commonly used on production websites. As the keyword represents the color as a human-readable text value, named colors are used in code examples to clearly tell the user what color is expected so the learner can focus on the content being taught.

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

antiquewhite

blueviolet

greenyellow

Interactive editor

```
.one {
  background-color: antiquewhite;
}

.two {
  background-color: blueviolet;
}

.three {
  background-color: greenyellow;
}
```

```
<div class="wrapper">
  <div class="box one">antiquewhite</div>
  <div class="box two">blueviolet</div>
  <div class="box three">greenyellow</div>
</div>
```

Reset

Hexadecimal RGB values

The next type of color value you are likely to encounter is hexadecimal codes. Hexadecimal uses 16 characters from `0–9` and `a–f`, so the entire range is `0123456789abcdef`. Each hex color value consists of a hash/pound symbol (`#`) followed by three or six hexadecimal characters (`#fcc` or `#ffc0cb`, for example), with an optional one

or two hexadecimal characters representing the alpha-transparency of the previous three or six character color values.

When using hexadecimal to describe RGB values, each **pair** of hexadecimal characters is a decimal number representing one of the channels — red, green and blue — and allows us to specify any of the 256 available values for each (16 x 16 = 256). These values are less intuitive than keywords for defining colors, but they are a lot more versatile because you can represent any RGB color with them.

#02798b

#c55da1

#128a7d

Interactive editor

```
.one {
  background-color: #02798b;
}

.two {
  background-color: #c55da1;
}

.three {
  background-color: #128a7d;
}
```

```
<div class="wrapper">
  <div class="box one">#02798b</div>
  <div class="box two">#c55da1</div>
  <div class="box three">#128a7d</div>
</div>
```

Reset

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

RGB values

To create RGB values directly, the `rgb()` function takes three parameters representing **red**, **green**, and **blue** channel values of the colors, with an optional fourth value separated by a slash ('/') representing opacity, 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 or a percent between 0% and 100% inclusive (but not a mixture of the two).

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

rgb(2 121 139)

rgb(197 93 161)

rgb(18 138 125)

Interactive editor

```
.one {
  background-color: rgb(2 121 139);
}

.two {
  background-color: rgb(197 93 161);
}

.three {
  background-color: rgb(18 138 125);
}
```

```
<div class="wrapper">
  <div class="box one">rgb(2 121 139)</div>
  <div class="box two">rgb(197 93 161)</div>
  <div class="box three">rgb(18 138 125)</div>
</div>
```

Reset

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.

rgb(2 121 139 / .3)

rgb(197 93 161 / .7)

rgb(18 138 125 / .9)

Interactive editor

```
.one {
  background-color: rgb(2 121 139 / .3);
}

.two {
  background-color: rgb(197 93 161 / .7);
}

.three {
  background-color: rgb(18 138 125 / .9);
}
```

```
<div class="wrapper">
  <div class="box one">rgb(2 121 139 / .3)</div>
  <div class="box two">rgb(197 93 161 / .7)</div>
  <div class="box three">rgb(18 138 125 / .9)</div>
</div>
```

Reset

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()`.

SRGB values

The `sRGB` color space defines colors in the **red** (r), **green** (g), and **blue** (b) color space.

Using hues to specify a color

If you want to go beyond keywords, hexadecimal, and `rgb()` for colors, you might want to try using `<hue>`. Hue is the property that allows us to tell the difference or similarity between colors like red, orange, yellow, green, blue, etc. The key concept is that you can specify a hue in an `<angle>` because most of the color models describe hues using a color wheel.

There are several color functions that include a `<hue>` component, including `hsl()`, `hwb()`, and `lch()`. Other color functions, like `lab()`, define colors based on what humans can see.

If you want to find out more about these functions and color spaces, see the Applying color to HTML elements using CSS guide, the `<color>` reference that lists all the different ways you can use colors in CSS, and the CSS color module that provides an overview of all the color types in CSS and the properties that use color values.

HWB

A great starting point for using hues in CSS is the `hwb()` function which specifies an `srgb()` color. The three parts are:

- **Hue**: The base shade of the color. This takes a `<hue>` value between 0 and 360, representing the angles around a color wheel.
- **Whiteness**: How white is the color? This takes a value from `0%` (no whiteness) to `100%` (full whiteness).

- **Blackness**: How black is the color? This takes a value from 0% (no blackness) to 100% (full blackness).

HSL

Similar to the `hwb()` function is the `hsl()` function which also specifies an `srgb()` color. HSL uses `Hue`, in addition to `Saturation` and `Lightness`:

- **Hue**
- **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).

The `hsl()` color value also has an optional fourth value, separated from the color with a slash (`/`), representing the alpha transparency.

Let's update the RGB example to use HSL colors instead:

hsl(188 97% 28%)

hsl(321 47% 57%)

hsl(174 77% 31%)

Interactive editor

```
.one {
  background-color: hsl(188 97% 28%);
}

.two {
  background-color: hsl(321 47% 57%);
}

.three {
  background-color: hsl(174 77% 31%);
}
```

```
<div class="wrapper">
  <div class="box one">hsl(188 97% 28%)</div>
  <div class="box two">hsl(321 47% 57%)</div>
  <div class="box three">hsl(174 77% 31%)</div>
</div>
```

Reset

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

hsl(188 97% 28% / .3)

hsl(321 47% 57% / .7)

hsl(174 77% 31% / .9)

Interactive editor

```
.one {
  background-color: hsl(188 97% 28% / .3);
}

.two {
  background-color: hsl(321 47% 57% / .7);
}

.three {
  background-color: hsl(174 77% 31% / .9);
}
```

```
<div class="wrapper">
  <div class="box one">hsl(188 97% 28% / .3)</div>
  <div class="box two">hsl(321 47% 57% / .7)</div>
  <div class="box three">hsl(174 77% 31% / .9)</div>
</div>
```

Reset

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()`.

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.

Interactive editor

```
.image {
  background-image: url(star.png);
}

.gradient {
  background-image: linear-gradient(90deg, rgb(119 0 255 / 39%),
rgb(0 212 255 / 100%));
}
```

```
<div class="box image"></div>
<div class="box gradient"></div>
```

Reset

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.

Interactive editor

```
.box {
  height: 300px;
  width: 400px;
  background-image: url(star.png);
  background-repeat: no-repeat;
  background-position: right 40px;
}
```

```
<div class="box"></div>
```

Reset

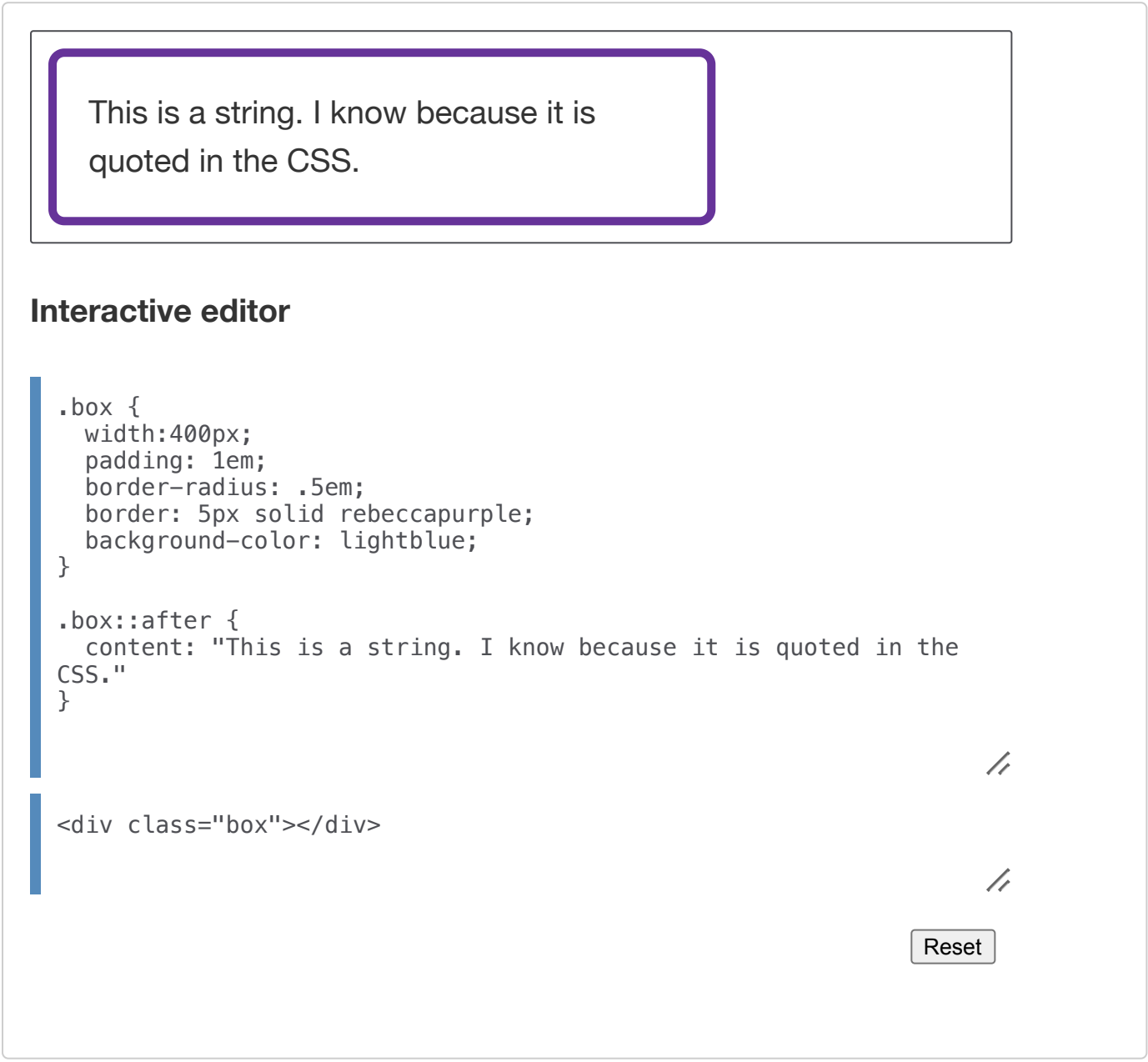
Play around with these values to see how you can push the image around.

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

In programming, a function is a piece of code that does a specific task. Functions are useful because you can write code once, then reuse it many times instead of writing the same logic over and over. Most programming languages not only support functions but

also come with convenient built-in functions for common tasks so you don't have to write them yourself from scratch.

CSS also has functions, which work in a similar way to functions in other languages. In fact, we've already seen CSS functions in the Color section above with `rgb()` and `hsl()` functions.

Aside from applying colors, you can use functions in CSS to do a lot of other things. For example Transform functions are a common way to move, rotate, and scale elements on a page. You might see `translate()` for moving something horizontally or vertically, `rotate()` to rotate something, or `scale()` to make something bigger or smaller.

Math functions

When you are creating styles for a project, you will probably start off with numbers like `300px` for lengths or `200ms` for durations. If you want to have these values change based on other values, you will need to do some math. You could calculate the percentage of a value or add a number to another number, then update your CSS with the result.

CSS has support for Math functions, which allow us to perform calculations instead of relying on static values or doing the math in JavaScript. One of the most common math functions is `calc()` which lets you do operations like addition, subtraction, multiplication, and division.

For example, let's say we want to set the width of an element to be 20% of its parent container plus 100px. We can't specify this width with a static value — if the parent uses a percentage width (or a relative unit like `em` or `rem`) then it will vary depending on the context it is used in, and other factors such as the user's device or browser window width. However, we can use `calc()` to set the width of the element to be 20% of its parent container plus 100px. The 20% is based on the width of the parent container (`.wrapper`) and if that width changes, the calculation will change too:

Interactive editor

```
.wrapper {
  width: 400px;
}

.box {
  width: calc(20% + 100px);
}
```

```
<div class="wrapper">
  <div class="box">My width is calculated.</div>
</div>
```

Reset

There are many other math functions that you can use in CSS, such as `min()`, `max()`, and `clamp()`; respectively these let you pick the smallest, largest, or middle value from a set of values. You can also use Trigonometric functions like `sin()`, `cos()`, and `tan()` to calculate angles for rotating elements around a point, or choose colors that take a hue angle as a parameter. Exponential functions might also be used for animations and transitions, when you require very specific control over how something moves and looks.

Knowing about CSS functions is useful so you recognize them when you see them. You should start experimenting with them in your projects — they will help you avoid writing custom or repetitive code to achieve results that you can get with regular CSS.

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.