

[HOME](#) / [GUIDES](#) /

A Complete Guide to Grid

**Chris House** May 12, 2021

Nov 8, 2021

Our comprehensive guide to CSS grid, focusing on all the settings both for the grid parent container and the grid child elements.

Table of contents

Part 1: [Introduction](https://css-tricks.com/snippets/css/complete-guide-grid/#introduction) (<https://css-tricks.com/snippets/css/complete-guide-grid/#introduction>)

Part 2: [Basics & Browser Support](#) (#basics-browser-support)

Part 3: [Important Terminology](#) (<https://css-tricks.com/snippets/css/complete-guide-grid/#important-terminology>)

Part 4: [Grid Properties](#) (<https://css-tricks.com/snippets/css/complete-guide-grid/#grid-properties>)

Part 5: [Special Units & Functions](#) (#special-units-functions)

Part 6: [Fluid Columns Snippet](#) (<https://css-tricks.com/snippets/css/complete-guide-grid/#fluid-columns-snippet>)

Part 7: [Animation](#) (<https://css-tricks.com/snippets/css/complete-guide-grid/#animation>)

Get the poster!

Reference this guide a lot? Pin a copy up on the office wall.

BUY POSTER ([HTTPS://CSS-TRICKS.COM/PRODUCT/CSS-GRID-POSTER/](https://css-tricks.com/product/css-grid-poster/))



(<https://css-tricks.com/product/css-grid-poster/>)

(#introduction) Introduction

CSS Grid Layout (aka “Grid” or “CSS Grid”), is a two-dimensional grid-based layout system that, compared to any web layout system of the past, completely changes the way we design user interfaces. CSS has always been used to layout our web pages, but it’s never done a very good job of it. First, we used tables, then floats, positioning and inline-block, but all of these methods were essentially hacks and left out a lot of important functionality (vertical centering, for instance). Flexbox (<https://css-tricks.com/snippets/css/a-guide-to-flexbox/>) is also a very great layout tool, but its one-directional flow has different use cases (<https://css-tricks.com/quick-whats-the-difference-between-flexbox-and-grid/>) — and they actually work together (<https://css-tricks.com/css-grid-replace-flexbox/>) quite well! Grid is the very

first CSS module created specifically to solve the layout problems we've all been hacking our way around for as long as we've been making websites.

The intention of this guide is to present the Grid concepts as they exist in the latest version of the specification. So I won't be covering the out-of-date Internet Explorer syntax (even though you can absolutely [use Grid in IE 11](https://css-tricks.com/css-grid-in-ie-debunking-common-ie-grid-misconceptions/) (<https://css-tricks.com/css-grid-in-ie-debunking-common-ie-grid-misconceptions/>)) or other historical hacks.

◀ [#basics-browser-support](#)) Basics & Browser Support

As of March 2017, most browsers shipped native, unprefixed support for CSS Grid: Chrome (including on Android), Firefox, Safari (including on iOS), and Opera. Internet Explorer 10 and 11 on the other hand support it, but it's an old implementation with an outdated syntax. The time to build with grid is now!

To get started you have to define a container element as a grid with [display: grid](#) (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-display>) , set the column and row sizes with [grid-template-columns](#) (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-template-columns-rows>) and [grid-template-rows](#) (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-template-columns-rows>) , and then place its child elements into the grid with [grid-column](#) (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-column-row>) and [grid-row](#) (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-column-row>) . Similarly to flexbox, the source order of the grid items doesn't matter. Your CSS can place them in any order, which makes it super easy to rearrange your grid with media queries. Imagine defining the layout of your entire page, and then completely rearranging it to accommodate a different screen width all with only a couple lines of CSS. Grid is one of the most powerful CSS modules ever introduced.

This browser support data is from Caniuse (<https://caniuse.com/#feat=css-grid>) , which has more detail. A number indicates that browser supports the feature at that version and up.

Desktop



Mobile / Tablet



⌚ #important-terminology) Important Terminology

Before diving into the concepts of Grid it's important to understand the terminology. Since the terms involved here are all kinda conceptually similar, it's easy to confuse them with one another if you don't first memorize their meanings defined by the Grid specification. But don't worry, there aren't many of them.

⌚ (#grid-container) Grid Container

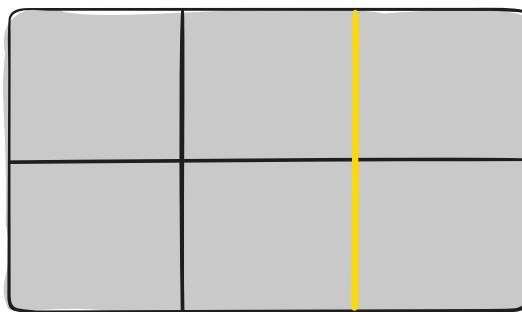
The element on which `display: grid` is applied. It's the direct parent of all the grid items. In this example `container` is the grid container.

```
<div class="container">
  <div class="item item-1"> </div>
  <div class="item item-2"> </div>
  <div class="item item-3"> </div>
</div>
```

HTML

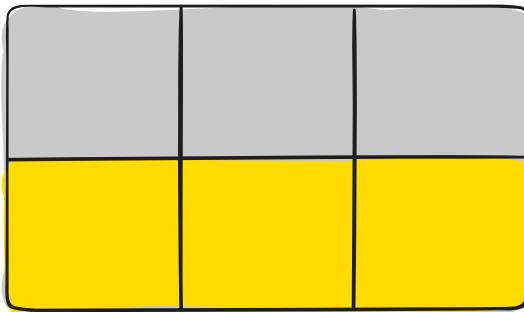
⌚ (#grid-line) (<https://css-tricks.com/snippets/css/complete-guide-grid/#grid-line>) Grid Line

The dividing lines that make up the structure of the grid. They can be either vertical (“column grid lines”) or horizontal (“row grid lines”) and reside on either side of a row or column. Here the yellow line is an example of a column grid line.



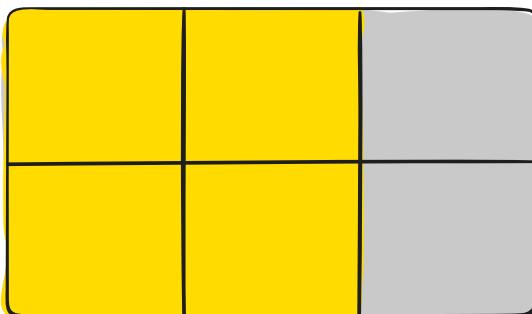
⌚ (#grid-track) (<https://css-tricks.com/snippets/css/complete-guide-grid/#grid-track>) Grid Track

The space between two adjacent grid lines. You can think of them as the columns or rows of the grid. Here's the grid track between the second and third-row grid lines.



⌚ (#grid-area) Grid Area

The total space surrounded by four grid lines. A grid area may be composed of any number of grid cells. Here's the grid area between row grid lines 1 and 3, and column grid lines 1 and 3.



⌚ (#grid-item) Grid Item

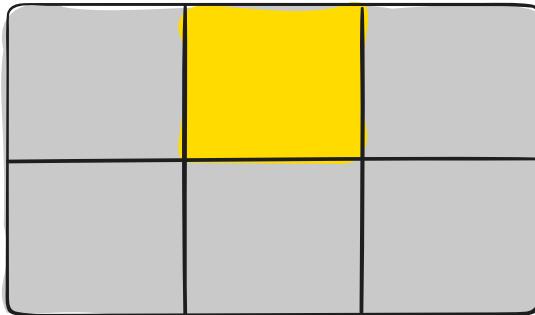
The children (i.e. *direct* descendants) of the grid container. Here the `item` elements are grid items, but `sub-item` isn't.

```
<div class="container">
  <div class="item" style="background-color: yellow; height: 100px; width: 100px; margin: 5px; border: 1px solid black;></div>
  <div class="item" style="background-color: yellow; height: 100px; width: 100px; margin: 5px; border: 1px solid black;></div>
  <div class="item" style="background-color: grey; height: 100px; width: 100px; margin: 5px; border: 1px solid black;></div>
</div>
```

HTML

⌚ (#grid-cell) (<https://css-tricks.com/snippets/css/complete-guide-grid/#grid-cell>) Grid Cell

The space between two adjacent row and two adjacent column grid lines. It's a single "unit" of the grid. Here's the grid cell between row grid lines 1 and 2, and column grid lines 2 and 3.



② #grid-properties **Grid Properties**

- ▶ Table of Contents of Grid Properties

② (#properties-for-the-parentgrid-container) **Properties for the Parent (Grid Container)**

② (#display) **display**

Defines the element as a grid container and establishes a new grid formatting context for its contents.

Values:

- **grid** – generates a block-level grid
- **inline-grid** – generates an inline-level grid

```
.container {  
  display: grid | inline-grid;  
}
```

css

Hey!

The ability to pass grid parameters down through nested elements (aka subgrids) has been moved to [level 2 of the CSS Grid specification](#). (<https://www.w3.org/TR/css-grid-2/#subgrids>) Here's a quick explanation (<https://css-tricks.com/grid-level-2-and-subgrid/>).

② (#grid-template-columns grid-template-rows) grid-template-columns grid-template-rows

Defines the columns and rows of the grid with a space-separated list of values. The values represent the track size, and the space between them represents the grid line.

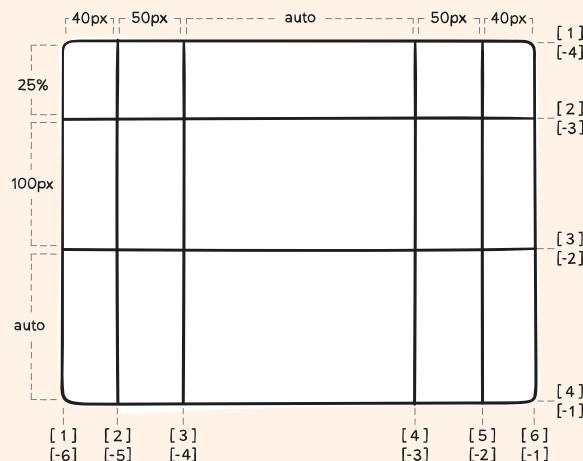
Values:

- **<track-size>** – can be a length, a percentage, or a fraction of the free space in the grid (using the [fr](https://css-tricks.com/introduction-fr-css-unit/) (<https://css-tricks.com/introduction-fr-css-unit/>) unit)
- **<line-name>** – an arbitrary name of your choosing

```
.container {
  grid-template-columns: ... ...;
  /* e.g.
   1fr 1fr
   minmax(10px, 1fr) 3fr
   repeat(5, 1fr)
   50px auto 100px 1fr
  */
  grid-template-rows: ... ...;
  /* e.g.
   min-content 1fr min-content
   100px 1fr max-content
  */
}
```

css

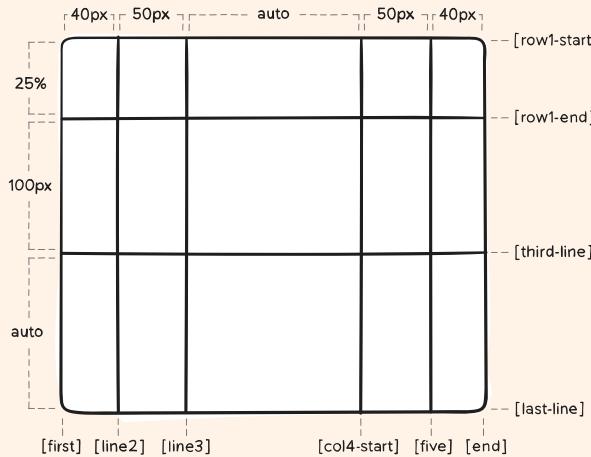
Grid lines are automatically assigned positive numbers from these assignments (-1 being an alternate for the very last row).



But you can choose to explicitly name the lines. Note the bracket syntax for the line names:

css

```
.container {
  grid-template-columns: [first] 40px [line2] 50px [line3] auto [col4-start] 50px [five] 40px [end];
  grid-template-rows: [row1-start] 25% [row1-end] 100px [third-line] auto [last-line];
}
```



Note that a line can have more than one name. For example, here the second line will have two names: row1-end and row2-start:

css

```
.container {
  grid-template-rows: [row1-start] 25% [row1-end row2-start] 25% [row2-end];
}
```

If your definition contains repeating parts, you can use the `repeat()` notation to streamline things:

css

```
.container {
  grid-template-columns: repeat(3, 20px [col-start]);
}
```

Which is equivalent to this:

css

```
.container {
  grid-template-columns: 20px [col-start] 20px [col-start] 20px [col-start];
}
```

If multiple lines share the same name, they can be referenced by their line name and count.

css

```
.item {
  grid-column-start: col-start 2;
}
```

The `fr` unit allows you to set the size of a track as a fraction of the free space of the grid container. For example, this will set each item to one third the width of the grid container:

css

```
.container {
  grid-template-columns: 1fr 1fr 1fr;
}
```

The free space is calculated *after* any non-flexible items. In this example the total amount of free space available to the `fr` units doesn't include the `50px`:

css

```
.container {
  grid-template-columns: 1fr 50px 1fr 1fr;
}
```

⌚ (#grid-template-areas) `grid-template-areas`

Defines a grid template by referencing the names of the grid areas which are specified with the `grid-area` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-area>) property. Repeating the name of a grid area causes the content to span those cells. A period signifies an empty cell. The syntax itself provides a visualization of the structure of the grid.

Values:

- `<grid-area-name>` – the name of a grid area specified with `grid-area` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-area>)
- `.` – a period signifies an empty grid cell
- `none` – no grid areas are defined

css

```
.container {
  grid-template-areas:
    "<grid-area-name> | . | none | ..."
    "...";
}
```

Example:

css

```
.item-a {
  grid-area: header;
}
.item-b {
  grid-area: main;
}
.item-c {
  grid-area: sidebar;
```

```

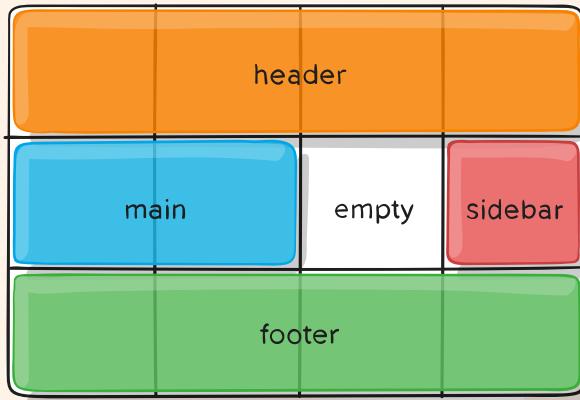
}

.item-d {
  grid-area: footer;
}

.container {
  display: grid;
  grid-template-columns: 50px 50px 50px 50px;
  grid-template-rows: auto;
  grid-template-areas:
    "header header header header"
    "main main . sidebar"
    "footer footer footer footer";
}

```

That'll create a grid that's four columns wide by three rows tall. The entire top row will be composed of the **header** area. The middle row will be composed of two **main** areas, one empty cell, and one **sidebar** area. The last row is all **footer**.



Each row in your declaration needs to have the same number of cells.

You can use any number of adjacent periods to declare a single empty cell. As long as the periods have no spaces between them they represent a single cell.

Notice that you're not naming lines with this syntax, just areas. When you use this syntax the lines on either end of the areas are actually getting named automatically. If the name of your grid area is **foo**, the name of the area's starting row line and starting column line will be **foo-start**, and the name of its last row line and last column line will be **foo-end**. This means that some lines might have multiple names, such as the far left line in the above example, which will have three names: header-start, main-start, and footer-start.

⌚ [\(#grid-template\) grid-template](#)

A shorthand for setting `grid-template-rows` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-template-columns-rows>) , `grid-template-columns` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-template-columns-rows>) , and `grid-template-areas` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-template-areas>) in a single declaration.

Values:

- **none** – sets all three properties to their initial values
- **<grid-template-rows> / <grid-template-columns>** – sets `grid-template-columns` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-template-columns-rows>) and `grid-template-rows` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-template-columns-rows>) to the specified values, respectively, and sets `grid-template-areas` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-template-areas>) to none

```
css
.container {
  grid-template: none | <grid-template-rows> / <grid-template-columns>;
}
```

It also accepts a more complex but quite handy syntax for specifying all three. Here's an example:

```
css
.container {
  grid-template:
    [row1-start] "header header header" 25px [row1-end]
    [row2-start] "footer footer footer" 25px [row2-end]
    / auto 50px auto;
}
```

That's equivalent to this:

```
css
.container {
  grid-template-rows: [row1-start] 25px [row1-end row2-start] 25px [row2-end];
  grid-template-columns: auto 50px auto;
  grid-template-areas:
    "header header header"
    "footer footer footer";
}
```

Since `grid-template` doesn't reset the *implicit* grid properties (`grid-auto-columns` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-auto-columns-rows>) , `grid-auto-rows` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-auto-columns-rows>) , and `grid-auto-flow` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-auto-columns-rows>))

[tricks.com/snippets/css/complete-guide-grid/#prop-grid-auto-flow](https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-auto-flow)), which is probably what you want to do in most cases, it's recommended to use the `grid` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid>) property instead of `grid-template`.

② (#column-gap row-gap grid-column-gap grid-row-gap) `column-gap` `row-gap` `grid-column-gap` `grid-row-gap`

Specifies the size of the grid lines. You can think of it like setting the width of the gutters between the columns/rows.

Values:

- `<line-size>` – a length value

```
.container {
  /* standard */
  column-gap: <line-size>;
  row-gap: <line-size>;

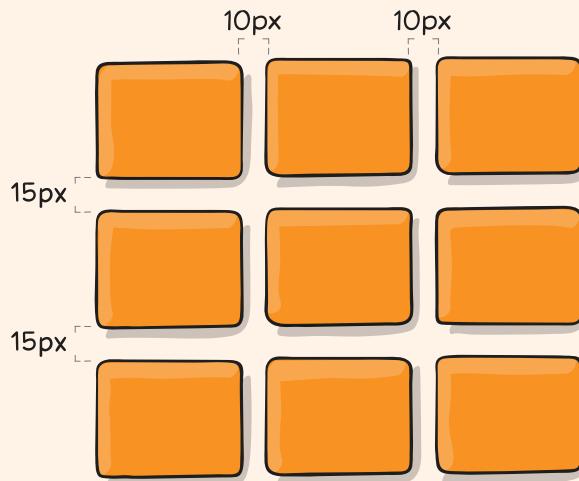
  /* old */
  grid-column-gap: <line-size>;
  grid-row-gap: <line-size>;
}
```

css

Example:

```
.container {
  grid-template-columns: 100px 50px 100px;
  grid-template-rows: 80px auto 80px;
  column-gap: 10px;
  row-gap: 15px;
}
```

css



The gutters are only created *between* the columns/rows, not on the outer edges.

Note: The `grid-` prefix will be removed and `grid-column-gap` and `grid-row-gap` renamed to `column-gap` and `row-gap`. The unprefixed properties are already supported in Chrome 68+, Safari 11.2 Release 50+, and Opera 54+.

⌚ (#gapgrid-gap) **gap** **grid-gap**

A shorthand for `row-gap` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-column-row-gap>) and `column-gap` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-column-row-gap>)

Values:

- **<grid-row-gap> <grid-column-gap>** – length values

```
.container {
  /* standard */
  gap: <grid-row-gap> <grid-column-gap>

  /* old */
  grid-gap: <grid-row-gap> <grid-column-gap>;
}
```

css

Example:

```
.container {
  grid-template-columns: 100px 50px 100px;
```

css

```
grid-template-rows: 80px auto 80px;
gap: 15px 10px;
}
```

If no `row-gap` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-column-row-gap>) is specified, it's set to the same value as `column-gap` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-column-row-gap>)

Note: The `grid-` prefix is deprecated (but who knows, may never actually be removed from browsers). Essentially `grid-gap` renamed to `gap`. The unprefixed property is already supported in Chrome 68+, Safari 11.2 Release 50+, and Opera 54+.

⌚ (#justify-items) justify-items

Aligns grid items along the *inline (row)* axis (as opposed to `align-items` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-align-items>)) which aligns along the *block (column)* axis). This value applies to all grid items inside the container.

Values:

- **start** – aligns items to be flush with the start edge of their cell
- **end** – aligns items to be flush with the end edge of their cell
- **center** – aligns items in the center of their cell
- **stretch** – fills the whole width of the cell (this is the default)

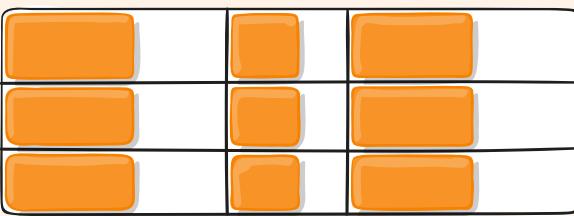
```
.container {
  justify-items: start | end | center | stretch;
}
```

css

Examples:

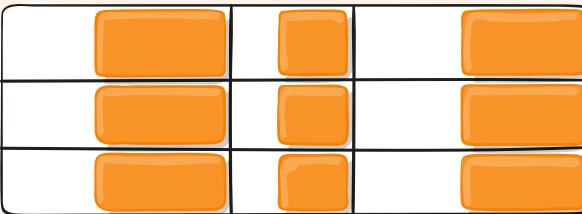
```
.container {
  justify-items: start;
}
```

css



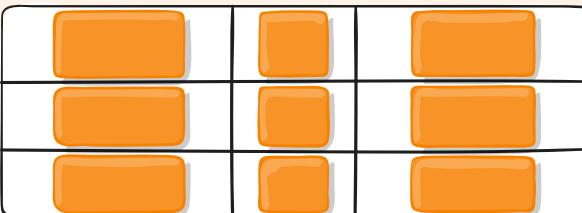
```
.container {  
  justify-items: end;  
}
```

css



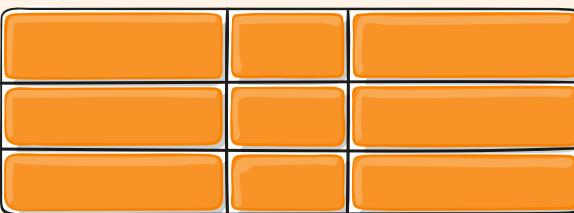
```
.container {  
  justify-items: center;  
}
```

css



```
.container {  
  justify-items: stretch;  
}
```

css



This behavior can also be set on individual grid items via the [justify-self](https://css-tricks.com/snippets/css/complete-guide-grid/#prop-justify-self) (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-justify-self>) property.

🔗 [\(#align-items\)](#) align-items

Aligns grid items along the *block (column)* axis (as opposed to [justify-items](#) (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-justify-items>) which aligns along the *inline (row)* axis). This value applies to all grid items inside the container.

Values:

- **stretch** – fills the whole height of the cell (this is the default)
- **start** – aligns items to be flush with the start edge of their cell
- **end** – aligns items to be flush with the end edge of their cell
- **center** – aligns items in the center of their cell
- **baseline** – align items along text baseline (<https://codepen.io/chriscoyier/pen/NWvvPRj>) ..

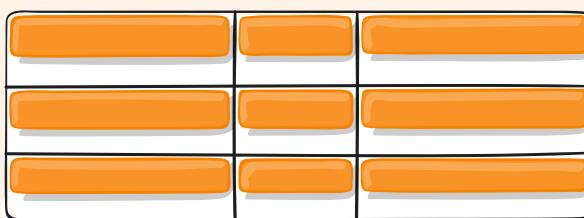
```
.container {
  align-items: start | end | center | stretch;
}
```

css

Examples:

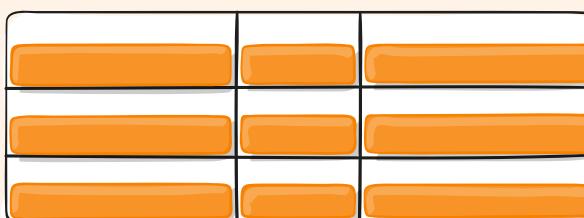
```
.container {
  align-items: start;
}
```

css

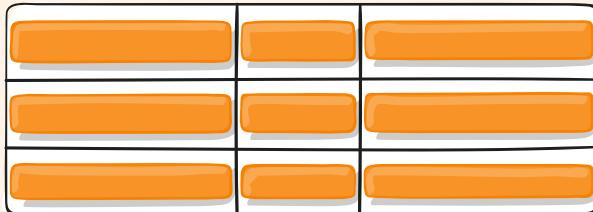


```
.container {
  align-items: end;
}
```

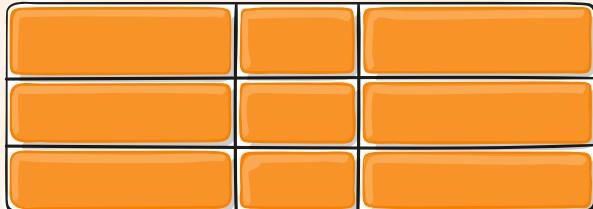
css



```
.container {
  align-items: center;
}
```



```
.container {
  align-items: stretch;
}
```



This behavior can also be set on individual grid items via the [align-self](https://css-tricks.com/snippets/css/complete-guide-grid/#prop-align-self) (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-align-self>) property.

☞ [#place-items](#) place-items

`place-items` sets both the `align-items` and `justify-items` properties in a single declaration.

Values:

- `<align-items>/<justify-items>` – The first value sets `align-items`, the second value `justify-items`. If the second value is omitted, the first value is assigned to both properties.

For more details, see [align-items](https://css-tricks.com/snippets/css/complete-guide-grid/#prop-align-items) (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-align-items>) and [justify-items](https://css-tricks.com/snippets/css/complete-guide-grid/#prop-justify-items) (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-justify-items>).

This can be very useful for super quick multi-directional centering:

```
.center {
  display: grid;
  place-items: center;
}
```

🔗 (#justify-content) **justify-content**

Sometimes the total size of your grid might be less than the size of its grid container. This could happen if all of your grid items are sized with non-flexible units like px. In this case you can set the alignment of the grid within the grid container. This property aligns the grid along the *inline (row)* axis (as opposed to [align-content](https://css-tricks.com/snippets/css/complete-guide-grid/#prop-align-content) (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-align-content>) which aligns the grid along the *block (column)* axis).

Values:

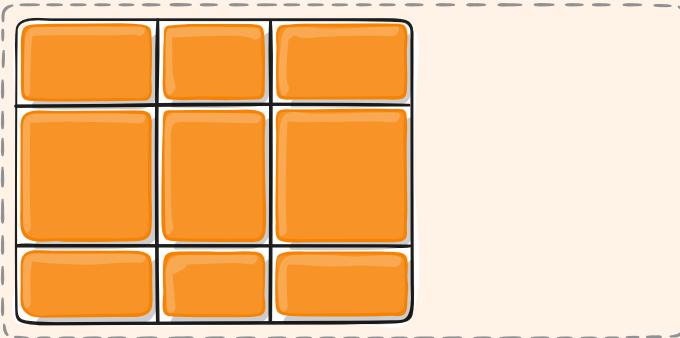
- **start** – aligns the grid to be flush with the start edge of the grid container
- **end** – aligns the grid to be flush with the end edge of the grid container
- **center** – aligns the grid in the center of the grid container
- **stretch** – resizes the grid items to allow the grid to fill the full width of the grid container
- **space-around** – places an even amount of space between each grid item, with half-sized spaces on the far ends
- **space-between** – places an even amount of space between each grid item, with no space at the far ends
- **space-evenly** – places an even amount of space between each grid item, including the far ends

```
.container {
  justify-content: start | end | center | stretch | space-around | space-between | space-evenly;
}
```

Examples:

```
.container {
  justify-content: start;
}
```

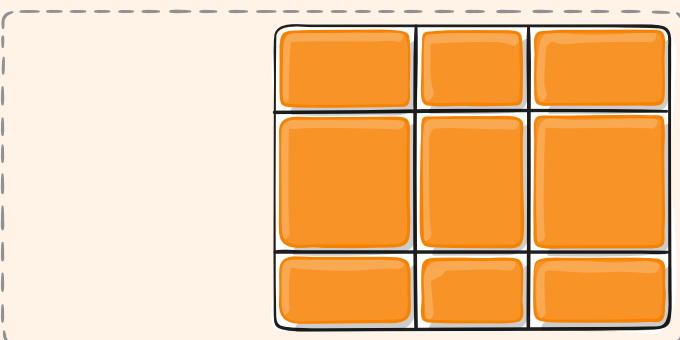
grid container



css

```
.container {  
  justify-content: end;  
}
```

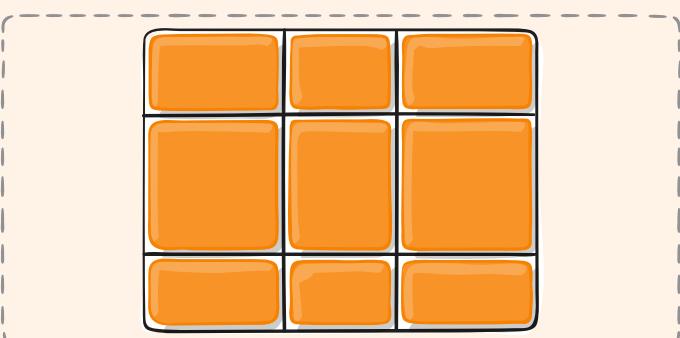
grid container



css

```
.container {  
  justify-content: center;  
}
```

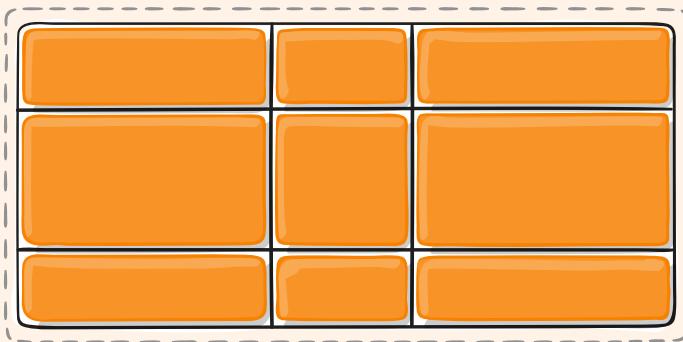
grid container



css

```
.container {  
  justify-content: stretch;  
}
```

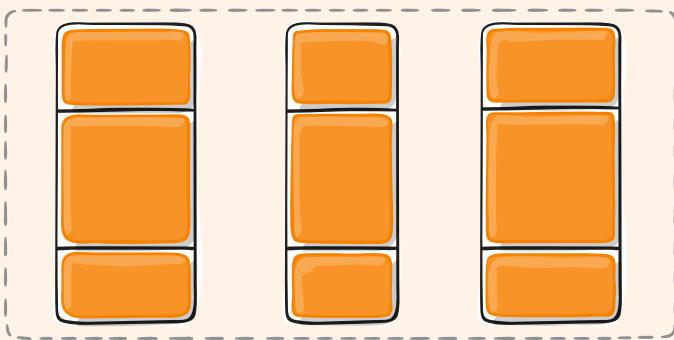
grid container



```
.container {  
  justify-content: space-around;  
}
```

css

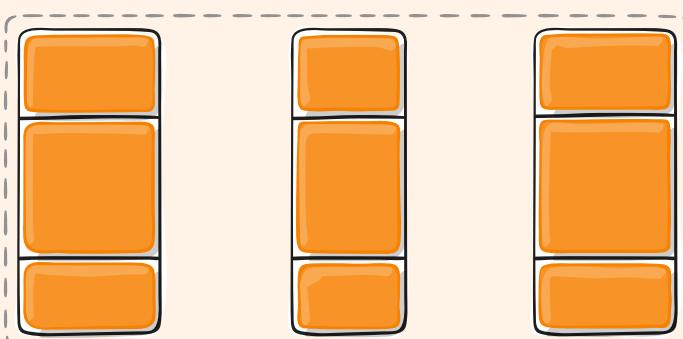
grid container



```
.container {  
  justify-content: space-between;  
}
```

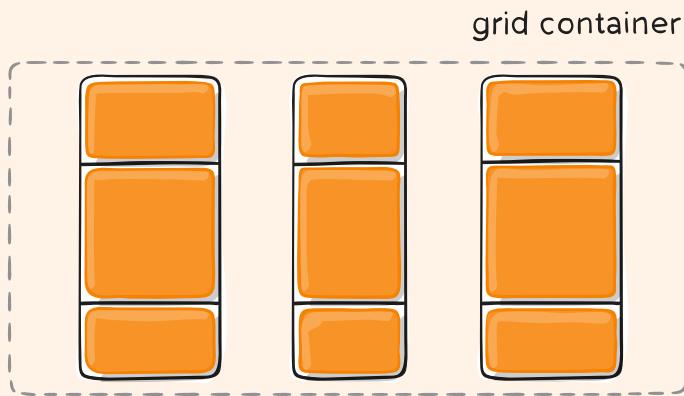
css

grid container



```
.container {  
  justify-content: space-evenly;  
}
```

css



⌚ (#align-content) align-content

Sometimes the total size of your grid might be less than the size of its grid container. This could happen if all of your grid items are sized with non-flexible units like px. In this case you can set the alignment of the grid within the grid container. This property aligns the grid along the *block (column)* axis (as opposed to [justify-content](#) (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-justify-content>)) which aligns the grid along the *inline (row)* axis).

Values:

- **start** – aligns the grid to be flush with the start edge of the grid container
- **end** – aligns the grid to be flush with the end edge of the grid container
- **center** – aligns the grid in the center of the grid container
- **stretch** – resizes the grid items to allow the grid to fill the full height of the grid container
- **space-around** – places an even amount of space between each grid item, with half-sized spaces on the far ends
- **space-between** – places an even amount of space between each grid item, with no space at the far ends
- **space-evenly** – places an even amount of space between each grid item, including the far ends

```
.container {
  align-content: start | end | center | stretch | space-around | space-between | space-evenly;
}
```

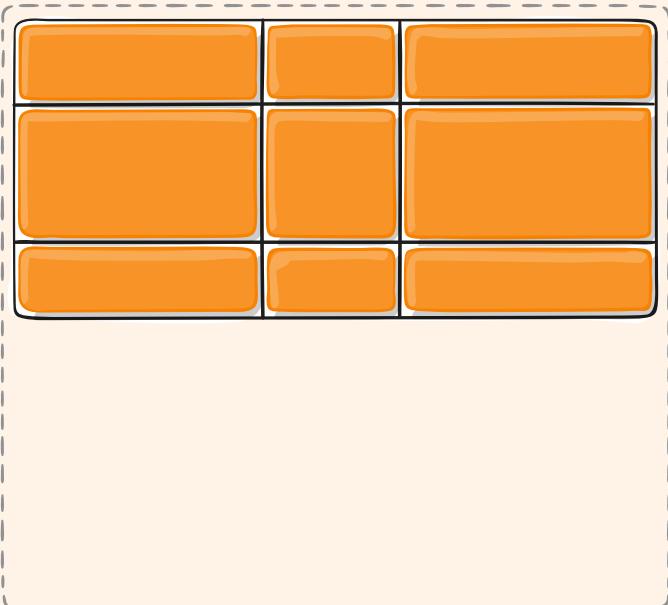
css

Examples:

css

```
.container {  
  align-content: start;  
}
```

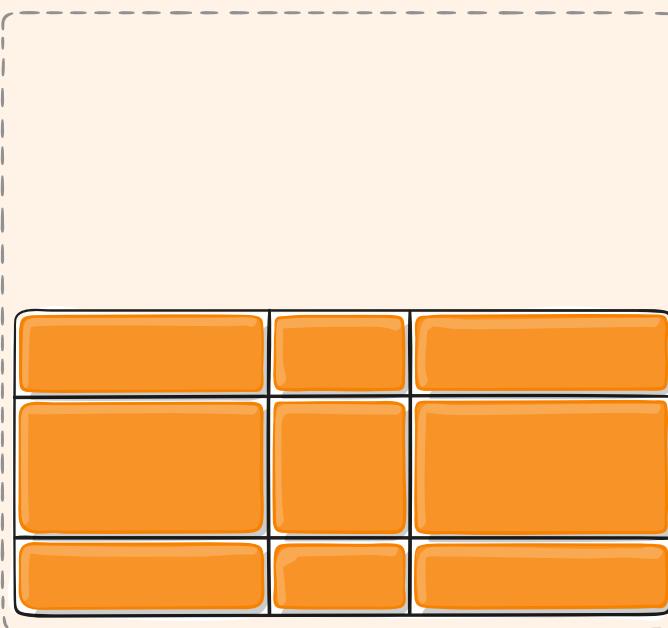
grid container



css

```
.container {  
  align-content: end;  
}
```

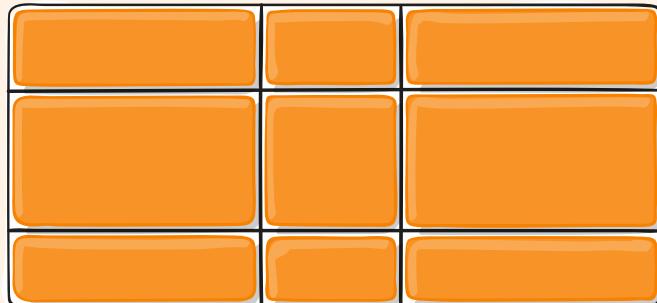
grid container



css

```
.container {  
  align-content: center;  
}
```

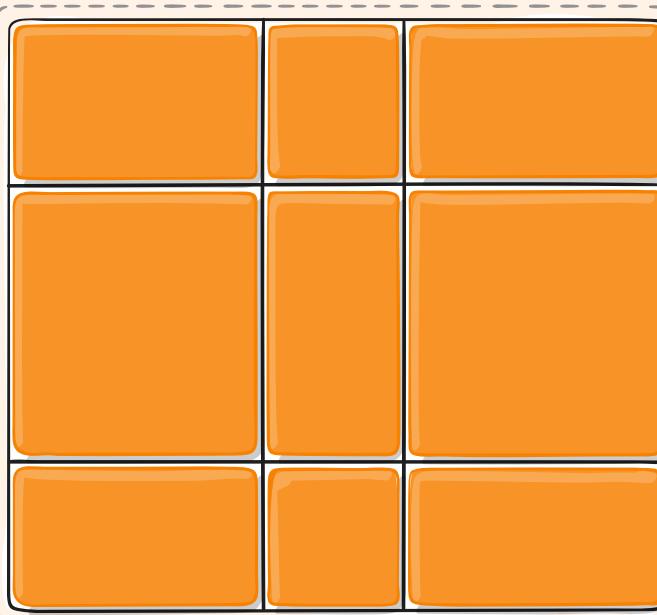
grid container



```
.container {  
  align-content: stretch;  
}
```

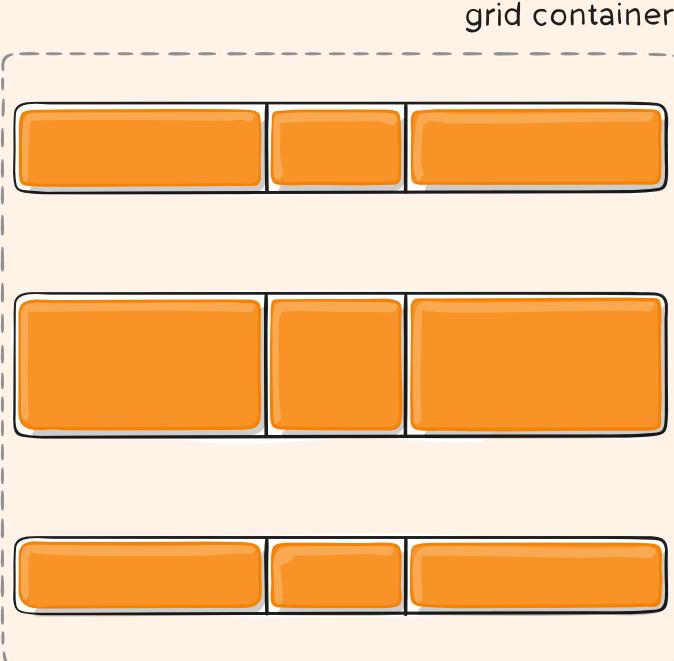
css

grid container



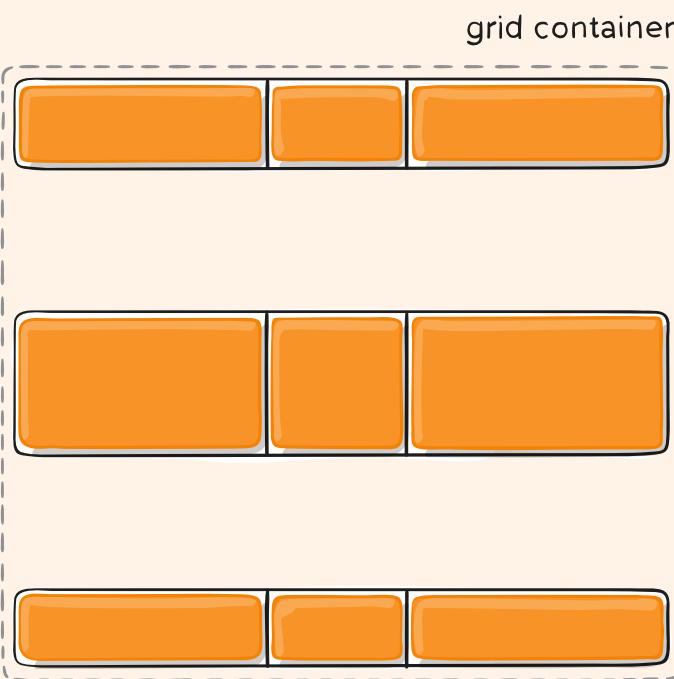
```
.container {  
  align-content: space-around;  
}
```

css



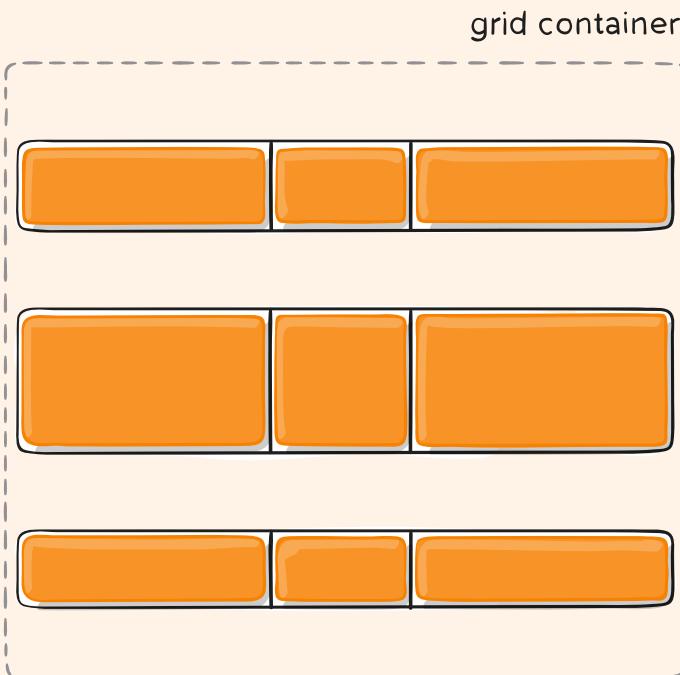
```
.container {  
  align-content: space-between;  
}
```

css



```
.container {  
  align-content: space-around;  
}
```

css



⌚ (#place-content) place-content

place-content sets both the align-content and justify-content properties in a single declaration.

Values:

- **<align-content> / <justify-content>** – The first value sets align-content, the second value justify-content. If the second value is omitted, the first value is assigned to both properties.

All major browsers except Edge support the place-content shorthand property.

For more details, see [align-content](https://css-tricks.com/snippets/css/complete-guide-grid/#prop-align-content) (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-align-content>) and [justify-content](https://css-tricks.com/snippets/css/complete-guide-grid/#prop-justify-content) (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-justify-content>).

⌚ (#grid-auto-columnsgrid-auto-rows) grid-auto-columns grid-auto-rows

Specifies the size of any auto-generated grid tracks (aka *implicit grid tracks*). Implicit tracks get created when there are more grid items than cells in the grid or when a grid item is placed outside of the explicit grid. (see [The Difference Between Explicit and Implicit Grids](https://css-tricks.com/difference-explicit-implicit-grids/) (<https://css-tricks.com/difference-explicit-implicit-grids/>))

Values:

- **<track-size>** – can be a length, a percentage, or a fraction of the free space in the grid (using the `fr` (<https://css-tricks.com/snippets/css/complete-guide-grid/#fr-unit>) unit)

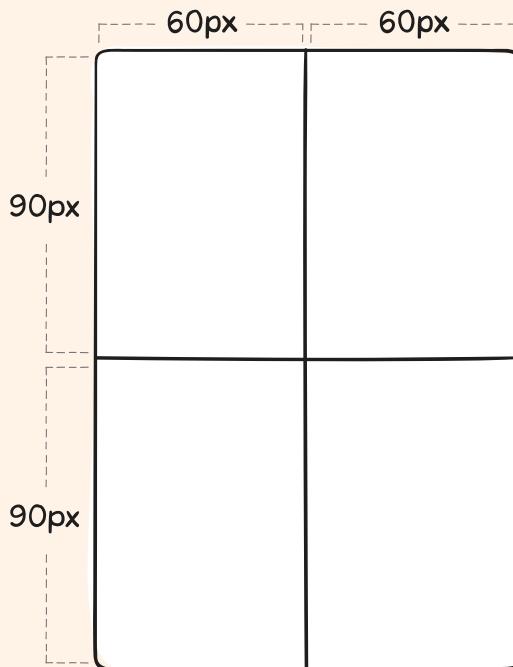
```
.container {
  grid-auto-columns: <track-size> ...;
  grid-auto-rows: <track-size> ...;
}
```

css

To illustrate how implicit grid tracks get created, think about this:

```
.container {
  grid-template-columns: 60px 60px;
  grid-template-rows: 90px 90px;
}
```

css



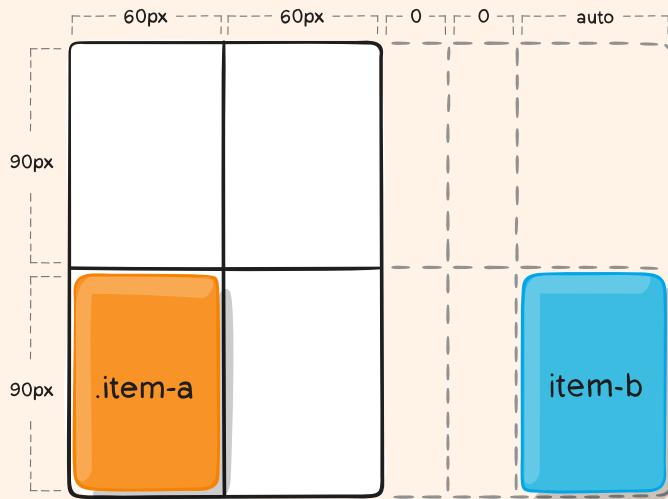
This creates a 2 x 2 grid.

But now imagine you use `grid-column` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-column-row>) and `grid-row` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-column-row>)

[tricks.com/snippets/css/complete-guide-grid/#prop-grid-column-row](https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-column-row) to position your grid items like this:

```
.item-a {
  grid-column: 1 / 2;
  grid-row: 2 / 3;
}
.item-b {
  grid-column: 5 / 6;
  grid-row: 2 / 3;
}
```

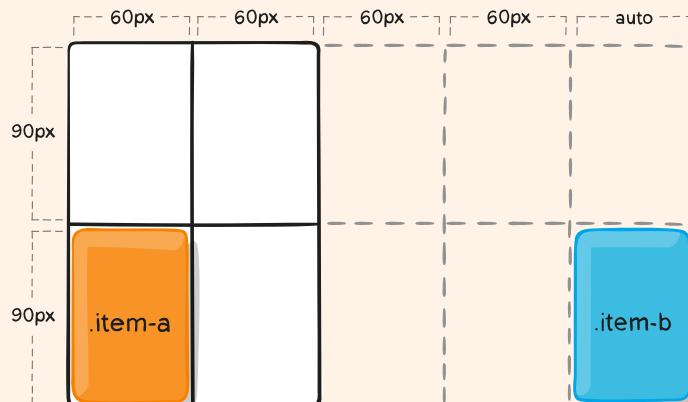
css



We told .item-b to start on column line 5 and end at column line 6, *but we never defined a column line 5 or 6*. Because we referenced lines that don't exist, implicit tracks with widths of 0 are created to fill in the gaps. We can use [grid-auto-columns](https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-auto-columns) (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-auto-columns-rows>) and [grid-auto-rows](https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-auto-rows) (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-auto-columns-rows>) to specify the widths of these implicit tracks:

```
.container {
  grid-auto-columns: 60px;
}
```

css



🔗 (#grid-auto-flow) **grid-auto-flow**

If you have grid items that you don't explicitly place on the grid, the *auto-placement algorithm* kicks in to automatically place the items. This property controls how the auto-placement algorithm works.

Values:

- **row** – tells the auto-placement algorithm to fill in each row in turn, adding new rows as necessary (default)
- **column** – tells the auto-placement algorithm to fill in each column in turn, adding new columns as necessary
- **dense** – tells the auto-placement algorithm to attempt to fill in holes earlier in the grid if smaller items come up later

```
.container {
  grid-auto-flow: row | column | row dense | column dense;
}
```

css

Note that **dense** only changes the visual order of your items and might cause them to appear out of order, which is bad for accessibility.

Examples:

Consider this HTML:

```
<section class="container">
  <div class="item-a">item-a</div>
  <div class="item-b">item-b</div>
  <div class="item-c">item-c</div>
  <div class="item-d">item-d</div>
  <div class="item-e">item-e</div>
</section>
```

html

You define a grid with five columns and two rows, and set `grid-auto-flow` to `row` (which is also the default):

```
.container {
  display: grid;
  grid-template-columns: 60px 60px 60px 60px 60px;
  grid-template-rows: 30px 30px;
```

css

```
grid-auto-flow: row;
}
```

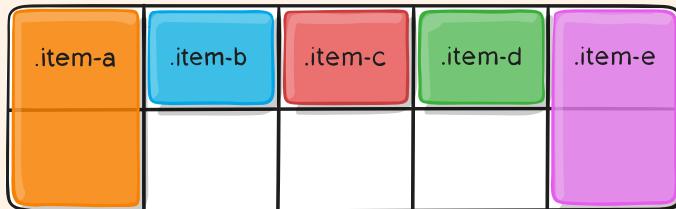
When placing the items on the grid, you only specify spots for two of them:

```
.item-a {
  grid-column: 1;
  grid-row: 1 / 3;
}

.item-e {
  grid-column: 5;
  grid-row: 1 / 3;
}
```

css

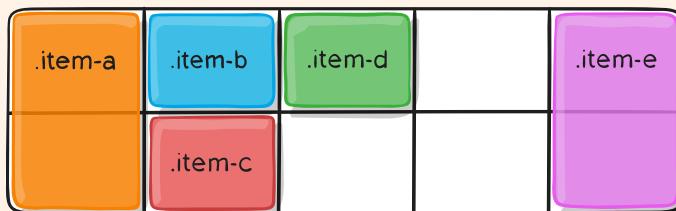
Because we set `grid-auto-flow` to `row`, our grid will look like this. Notice how the three items we didn't place (**item-b**, **item-c** and **item-d**) flow across the available rows:



If we instead set `grid-auto-flow` to `column`, **item-b**, **item-c** and **item-d** flow down the columns:

```
.container {
  display: grid;
  grid-template-columns: 60px 60px 60px 60px 60px;
  grid-template-rows: 30px 30px;
  grid-auto-flow: column;
}
```

css



↻ [#grid](#) **grid**

A shorthand for setting all of the following properties in a single declaration: `grid-template-rows` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-template-columns-rows>) , `grid-template-columns` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-template-columns>) , `grid-template-areas` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-template-areas>) , `grid-auto-rows` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-auto-columns-rows>) , `grid-auto-columns` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-auto-columns>) , and `grid-auto-flow` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-auto-flow>) (Note: You can only specify the explicit or the implicit grid properties in a single grid declaration).

Values:

- `none` – sets all sub-properties to their initial values.
- `<grid-template>` – works the same as the `grid-template` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-template>) shorthand.
- `<grid-template-rows> / [auto-flow && dense?] <grid-auto-columns>?` – sets `grid-template-rows` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-template-columns-rows>) to the specified value. If the `auto-flow` keyword is to the right of the slash, it sets `grid-auto-flow` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-auto-flow>) to column. If the `dense` keyword is specified additionally, the auto-placement algorithm uses a “dense” packing algorithm. If `grid-auto-columns` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-auto-columns-rows>) is omitted, it is set to `auto`.
- `[auto-flow && dense?] <grid-auto-rows>? / <grid-template-columns>?` – sets `grid-template-columns` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-template-columns-rows>) to the specified value. If the `auto-flow` keyword is to the left of the slash, it sets `grid-auto-flow` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-auto-flow>) to row. If the `dense` keyword is specified additionally, the auto-placement algorithm uses a “dense” packing algorithm. If `grid-auto-rows` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-auto-columns-rows>) is omitted, it is set to `auto`.

Examples:

The following two code blocks are equivalent:

```
.container {
  grid: 100px 300px / 3fr 1fr;
}
```

css

```
.container {
  grid-template-rows: 100px 300px;
  grid-template-columns: 3fr 1fr;
}
```

The following two code blocks are equivalent:

```
.container {
  grid: auto-flow / 200px 1fr;
}

.container {
  grid-auto-flow: row;
  grid-template-columns: 200px 1fr;
}
```

The following two code blocks are equivalent:

```
.container {
  grid: auto-flow dense 100px / 1fr 2fr;
}

.container {
  grid-auto-flow: row dense;
  grid-auto-rows: 100px;
  grid-template-columns: 1fr 2fr;
}
```

And the following two code blocks are equivalent:

```
.container {
  grid: 100px 300px / auto-flow 200px;
}

.container {
  grid-template-rows: 100px 300px;
  grid-auto-flow: column;
  grid-auto-columns: 200px;
}
```

It also accepts a more complex but quite handy syntax for setting everything at once. You specify `grid-template-areas` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-template-areas>) , `grid-template-rows` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-template-rows>) and `grid-template-columns` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-template-columns>) , and all the other sub-properties are set to their initial values. What you're doing is specifying the line names and track sizes inline with their respective grid areas. This is easiest to describe with an example:

```
.container {
  grid: [row1-start] "header header header" 1fr [row1-end]
        [row2-start] "footer footer footer" 25px [row2-end]
        / auto 50px auto;
}
```

css

That's equivalent to this:

```
.container {
  grid-template-areas:
    "header header header"
    "footer footer footer";
  grid-template-rows: [row1-start] 1fr [row1-end row2-start] 25px [row2-end];
  grid-template-columns: auto 50px auto;
}
```

css

② (#properties-for-the-childrengrid-items) Properties for the Children (Grid Items)

Hey!

`float, display: inline-block, display: table-cell, vertical-align and column-* properties have no effect on a grid item.`

② (#grid-column-startgrid-column-endgrid-row-startgrid-row-end) **grid-column-start**
grid-column-end
grid-row-start
grid-row-end

Determines a grid item's location within the grid by referring to specific grid lines. `grid-column-start/grid-row-start` is the line where the item begins, and `grid-column-end/grid-row-end` is the line where the item ends.

Values:

- **<line>** – can be a number to refer to a numbered grid line, or a name to refer to a named grid line
- **span <number>** – the item will span across the provided number of grid tracks
- **span <name>** – the item will span across until it hits the next line with the provided name

- **auto** – indicates auto-placement, an automatic span, or a default span of one

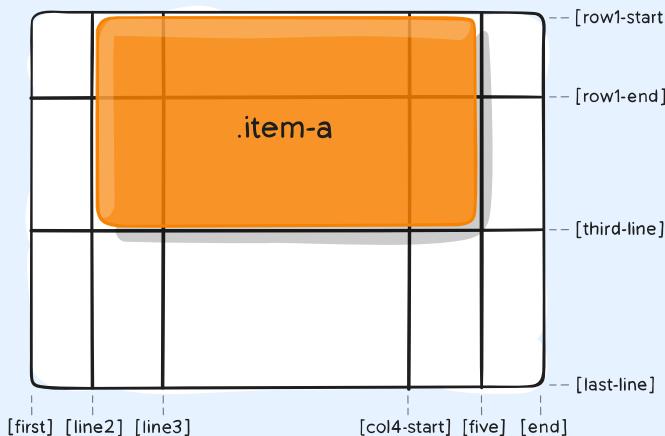
```
.item {
  grid-column-start: <number> | <name> | span <number> | span <name> | auto;
  grid-column-end: <number> | <name> | span <number> | span <name> | auto;
  grid-row-start: <number> | <name> | span <number> | span <name> | auto;
  grid-row-end: <number> | <name> | span <number> | span <name> | auto;
}
```

css

Examples:

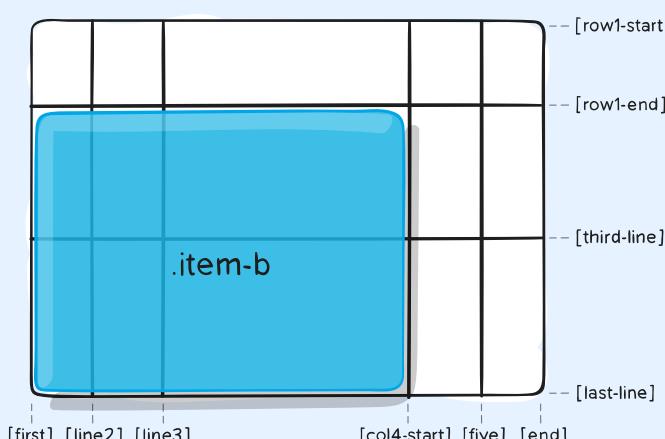
```
.item-a {
  grid-column-start: 2;
  grid-column-end: five;
  grid-row-start: row1-start;
  grid-row-end: 3;
}
```

css



```
.item-b {
  grid-column-start: 1;
  grid-column-end: span col4-start;
  grid-row-start: 2;
  grid-row-end: span 2;
}
```

css



If no `grid-column-end`/`grid-row-end` is declared, the item will span 1 track by default.

Items can overlap each other. You can use `z-index` to control their stacking order.

⌚ (#grid-columngrid-row) `grid-column` `grid-row`

Shorthand for `grid-column-start` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-column-row-start-end>) + `grid-column-end` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-column-row-start-end>) , and `grid-row-start` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-column-row-start-end>) + `grid-row-end` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-column-row-start-end>) , respectively.

Values:

- `<start-line>` / `<end-line>` – each one accepts all the same values as the longhand version, including span

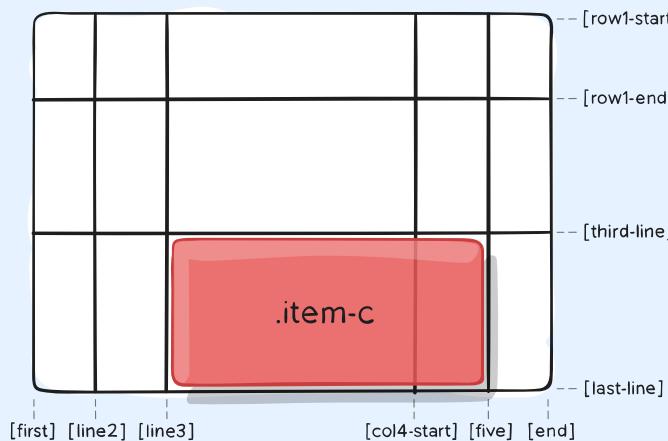
```
.item {
  grid-column: <start-line> / <end-line> | <start-line> / span <value>;
  grid-row: <start-line> / <end-line> | <start-line> / span <value>;
}
```

css

Example:

```
.item-c {
  grid-column: 3 / span 2;
  grid-row: third-line / 4;
}
```

css



If no end line value is declared, the item will span 1 track by default.

🔗 [\(#grid-area\)](#) `grid-area`

Gives an item a name so that it can be referenced by a template created with the `grid-template-areas` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-template-areas>) property. Alternatively, this property can be used as an even shorter shorthand for `grid-row-start` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-column-row-start-end>) + `grid-column-start` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-column-row-start-end>) + `grid-row-end` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-column-row-start-end>) + `grid-column-end` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-column-row-start-end>) .

Values:

- `<name>` – a name of your choosing
- `<row-start> / <column-start> / <row-end> / <column-end>` – can be numbers or named lines

```
.item {
  grid-area: <name> | <row-start> / <column-start> / <row-end> / <column-end>;
}
```

css

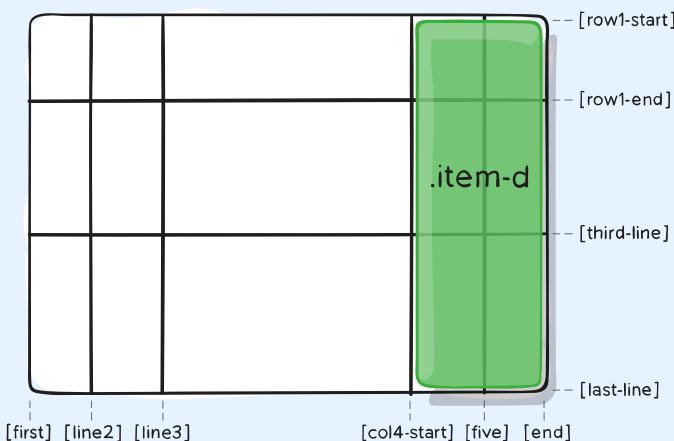
Examples:

As a way to assign a name to the item:

```
.item-d {
  grid-area: header;
}
```

As the short-shorthand for `grid-row-start` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-column-row-start-end>) + `grid-column-start` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-column-row-start-end>) + `grid-row-end` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-column-row-start-end>) + `grid-column-end` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-grid-column-row-start-end>) :

```
.item-d {
  grid-area: 1 / col4-start / last-line / 6;
}
```



⌚ `(#justify-self)` `justify-self`

Aligns a grid item inside a cell along the *inline (row)* axis (as opposed to `align-self` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-align-self>)) which aligns along the *block (column)* axis). This value applies to a grid item inside a single cell.

Values:

- **start** – aligns the grid item to be flush with the start edge of the cell
- **end** – aligns the grid item to be flush with the end edge of the cell
- **center** – aligns the grid item in the center of the cell
- **stretch** – fills the whole width of the cell (this is the default)

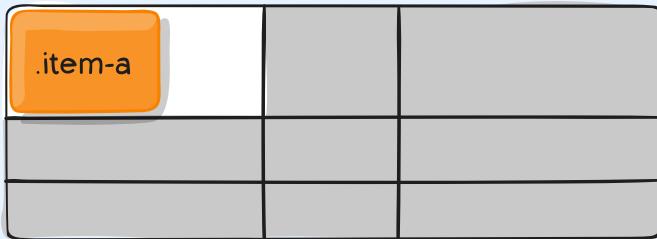
css

```
.item {  
  justify-self: start | end | center | stretch;  
}
```

Examples:

css

```
.item-a {  
  justify-self: start;  
}
```



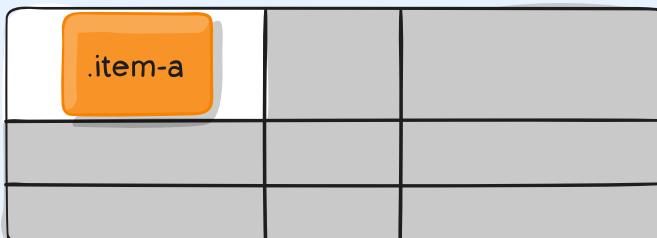
css

```
.item-a {  
  justify-self: end;  
}
```



css

```
.item-a {  
  justify-self: center;  
}
```



css

```
.item-a {  
  justify-self: stretch;  
}
```



To set alignment for *all* the items in a grid, this behavior can also be set on the grid container via the [justify-items](https://css-tricks.com/snippets/css/complete-guide-grid/#prop-justify-items) (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-justify-items>) property.

⌚ [\(#align-self\)](#) align-self

Aligns a grid item inside a cell along the *block* (*column*) axis (as opposed to [justify-self](https://css-tricks.com/snippets/css/complete-guide-grid/#prop-justify-self) (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-justify-self>)) which aligns along the *inline* (*row*) axis). This value applies to the content inside a single grid item.

Values:

- **start** – aligns the grid item to be flush with the start edge of the cell
- **end** – aligns the grid item to be flush with the end edge of the cell
- **center** – aligns the grid item in the center of the cell
- **stretch** – fills the whole height of the cell (this is the default)

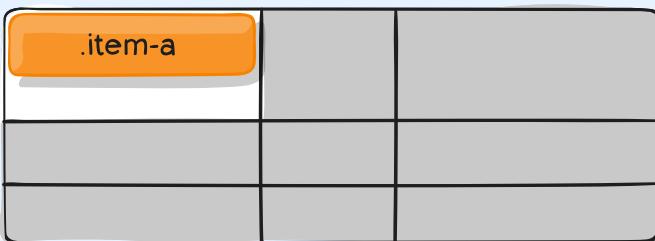
```
.item {
  align-self: start | end | center | stretch;
}
```

css

Examples:

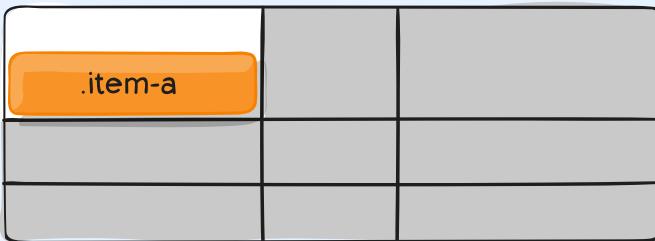
```
.item-a {
  align-self: start;
}
```

css



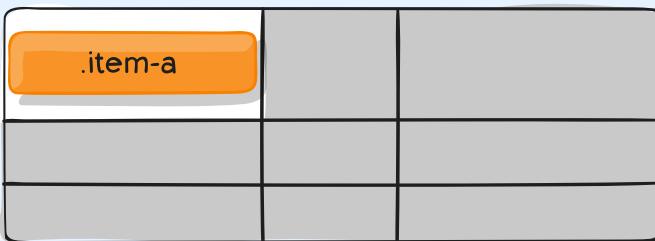
```
.item-a {  
  align-self: end;  
}
```

css



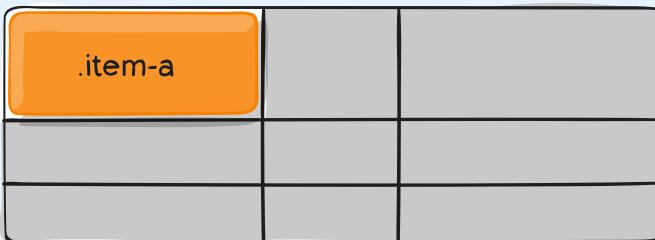
```
.item-a {  
  align-self: center;  
}
```

css



```
.item-a {  
  align-self: stretch;  
}
```

css



To align *all* the items in a grid, this behavior can also be set on the grid container via the `align-items` (<https://css-tricks.com/snippets/css/complete-guide-grid/#prop-align-items>) property.

🔗 (#place-self) place-self

place-self sets both the align-self and justify-self properties in a single declaration.

Values:

- **auto** – The “default” alignment for the layout mode.
- **<align-self> / <justify-self>** – The first value sets align-self, the second value justify-self. If the second value is omitted, the first value is assigned to both properties.

Examples:

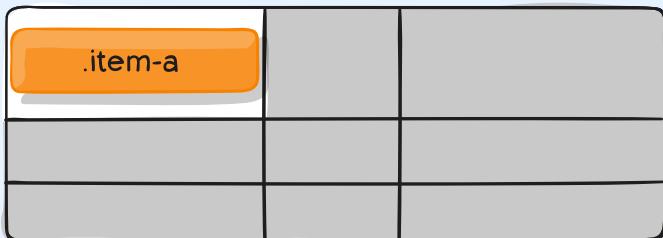
```
.item-a {  
  place-self: center;  
}
```

css



```
.item-a {  
  place-self: center stretch;  
}
```

css



All major browsers except Edge support the place-self shorthand property.

🔗 (#special-units-functions) Special Units & Functions

⌚ (#fr-units) fr units

You'll likely end up using a lot of [fractional units](https://css-tricks.com/introduction-fr-css-unit/) (<https://css-tricks.com/introduction-fr-css-unit/>) in CSS Grid, like `1fr`. They essentially mean "portion of the remaining space". So a declaration like:

```
grid-template-columns: 1fr 3fr;
```

css

Means, loosely, 25% 75%. Except that those percentage values are much more firm than fractional units are. For example, if you added padding to those percentage-based columns, now you've broken 100% width (assuming a `content-box` box model). Fractional units also much more friendly in combination with other units, as you can imagine:

```
grid-template-columns: 50px min-content 1fr;
```

css

⌚ (#sizing-keywords) Sizing Keywords

When sizing rows and columns, you can use all the [lengths](https://css-tricks.com/the-lengths-of-css/) (<https://css-tricks.com/the-lengths-of-css/>) you are used to, like `px`, `rem`, `%`, etc, but you also have keywords:

- `min-content`: the minimum size of the content. Imagine a line of text like "The very long hotdog.", the `min-content` is likely the width of the word "The".
- `max-content`: the maximum size of the content. Imagine the sentence above, the `max-content` is the length of the whole sentence.
- `auto`: this keyword is a lot like `fr` units, except that they "lose" the fight in sizing against `fr` units when allocating the remaining space.
- `fit-content`: use the space available, but never less than `min-content` and never more than `max-content`.
- `fractional units`: see above

⌚ (#sizing-functions) Sizing Functions

- The `minmax()` function does exactly what it seems like: it sets a minimum and maximum value for what the length is able to be. This is useful for in combination with relative units. Like you may want a column to be only able to shrink so far. This is [extremely useful and probably what you want](https://css-tricks.com/you-want-minmax10px-1fr-not-1fr/) (<https://css-tricks.com/you-want-minmax10px-1fr-not-1fr/>) :

```
grid-template-columns: minmax(100px, 1fr) 3fr;
```

css

- The `min()` function.
- The `max()` function.

🔗 (#the-repeat-function-and-keywords) The `repeat()` Function and Keywords

The `repeat()` function can save some typing:

```
grid-template-columns:  
 1fr 1fr 1fr 1fr 1fr 1fr 1fr 1fr;  
  
/* easier: */  
grid-template-columns:  
  repeat(8, 1fr);  
  
/* especially when: */  
grid-template-columns:  
  repeat(8, minmax(10px, 1fr));
```

css

But `repeat()` can get extra fancy when combined with keywords:

- `auto-fill`: Fit as many possible columns as possible on a row, even if they are empty.
- `auto-fit`: Fit whatever columns there are into the space. Prefer expanding columns to fill space rather than empty columns.

This bears the most famous snippet in all of CSS Grid and one of the all-time great CSS tricks (<https://css-tricks.com/books/greatest-css-tricks/flexible-grids/>) :

```
grid-template-columns:  
  repeat(auto-fit, minmax(250px, 1fr));
```

css

The difference between the keywords is [spelled out in detail here](https://css-tricks.com/auto-sizing-columns-css-grid-auto-fill-vs-auto-fit/) (<https://css-tricks.com/auto-sizing-columns-css-grid-auto-fill-vs-auto-fit/>).

⌚ (#masonry) Masonry

An experimental feature of CSS grid is masonry layout. Note that there are lots of approaches to CSS masonry (<https://css-tricks.com/piecing-together-approaches-for-a-css-masonry-layout/>) , but mostly of them are trickery and either have major downsides or aren't what you quite expect.

The spec has an official way (<https://drafts.csswg.org/css-grid-3/#masonry-layout>) now, and this is behind a flag in Firefox:

```
.container {  
  display: grid;  
  grid-template-columns: repeat(4, 1fr);  
  grid-template-rows: masonry;  
}
```

css

See Rachel's article (<https://www.smashingmagazine.com/native-css-masonry-layout-css-grid/>) for a deep dive.

⌚ (#subgrid) Subgrid

Subgrid is an extremely useful feature of grids that allows grid items to have a grid of their own that inherits grid lines from the parent grid.

```
.parent-grid {  
  display: grid;  
  grid-template-columns: repeat(9, 1fr);  
}  
.grid-item {  
  grid-column: 2 / 7;  
  
  display: grid;  
  grid-template-columns: subgrid;  
}  
.child-of-grid-item {  
  /* gets to participate on parent grid! */  
  grid-column: 3 / 6;  
}
```

css

This is only supported in Firefox (https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Grid_Layout/Subgrid#browser_compatibility) right now, but it really needs (<http://meyerweb.com/eric/thoughts/2016/01/15/subgrids-considered-essential/>) to get everywhere.

It's also useful to know about `display: contents;`. This is *not* the same as `subgrid`, but it can be a useful tool sometimes in a similar fashion.

```
<div class="grid-parent">
```

HTML

```
  <div class="grid-item"></div>
  <div class="grid-item"></div>

  <ul style="display: contents;">
    <!-- These grid-items get to participate on
        the same grid!-->
    <li class="grid-item"></li>
    <li class="grid-item"></li>
  </ul>

</div>
```

⤓ (#fluid-columns-snippet) Fluid Columns Snippet

Fluid width columns that break into more or less columns as space is available, with no media queries!

```
.grid {
  display: grid;
  grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));
  /* This is better for small screens, once min() is better supported */
  /* grid-template-columns: repeat(auto-fill, minmax(min(200px, 100%), 1fr)); */
  gap: 1rem;
}
```

css

Embedded Pen Here

⤓ (#animation) Animation

