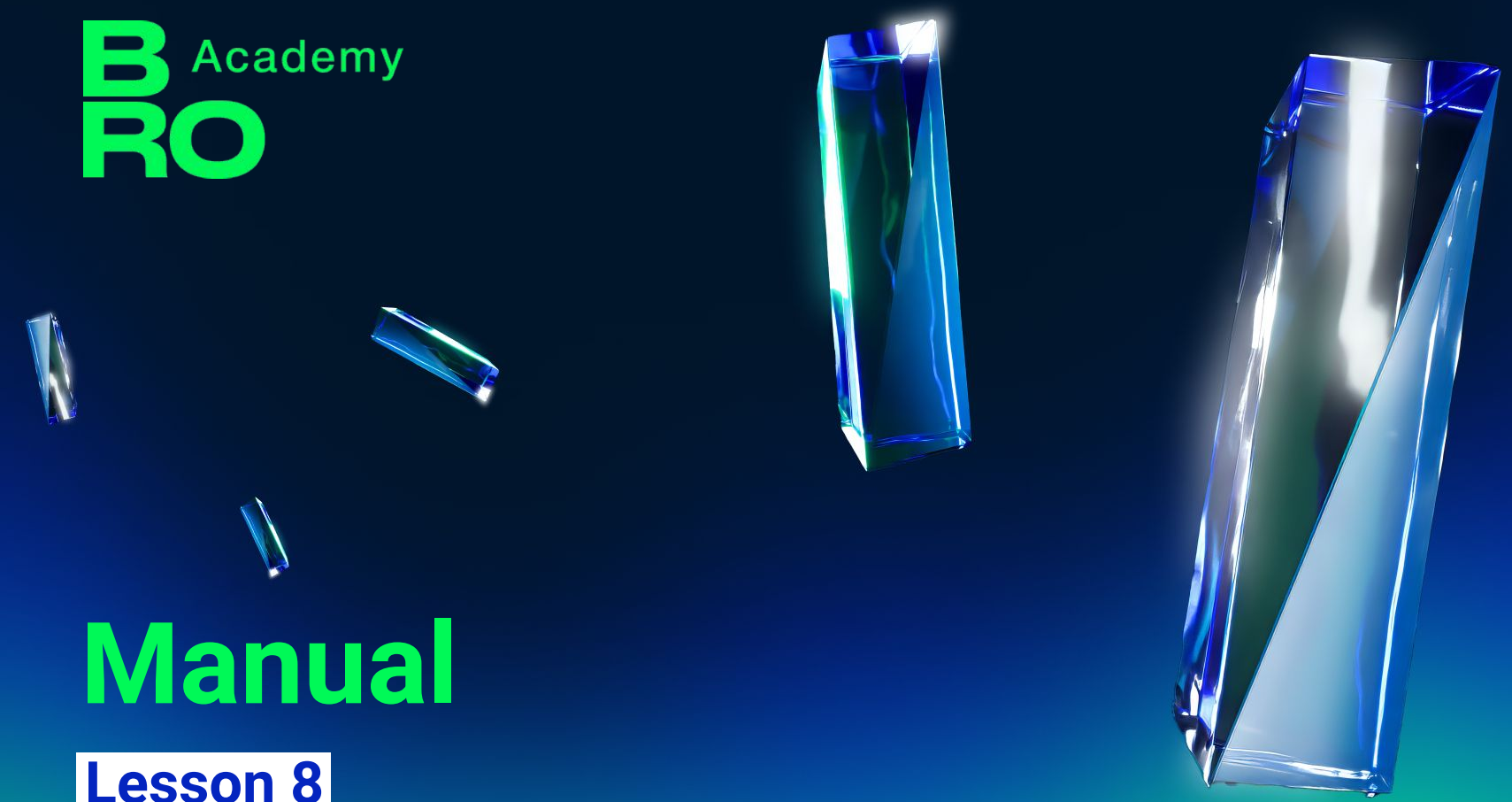
BRO Academy

Manual

Lesson 8



Grid

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 is also a very great layout tool, but its one-directional flow has different use cases — and they actually work together 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.

Flex vs grid

1. Flexbox can optionally wrap. If we allow a flex container to wrap, they will wrap down onto another row when the flex items fill a row. Where they line up on the next row is independent of what happened on the first row, allowing for a masonry-like look.
Grid can also optionally wrap (if we allow auto filling) in the sense that items can fill a row and move to the new row (or auto place themselves), but as they do, they will fall along the same grid lines all the other elements do.

2. Flexbox is essentially for laying out items in a single dimension – in a row OR a column. Grid is for layout of items in two dimensions – rows AND columns.

One-dimensional layouts
If you need to lay your content in one dimension, no matter if it's horizontal (x-axis) or vertical (y-axis) and you want your layout to respond to the element's content, then flexbox is what you're looking for. It's just right for (but not limited to) components.

Two-dimensional layouts
If you need to display your content both on the x-axis and y-axis that means you have... A grid! This property is ideal for page layouts and complicated, non-linear components.

3. **Grid is mostly defined on the parent element. In flexbox, most of the layout (beyond the very basics) happen on the children.**

4. **Grid is better at overlapping.** Getting elements to overlap in flexbox requires looking at traditional stuff, like negative margins, transforms, or absolute positioning in order to break out of the flex behavior. With grid, we can place items on overlapping grid lines, or even right within the same exact grid cells.

5. **Flexbox can push things away.** It's a rather unique feature of flexbox that you can, for example, put margin-right: auto; on an element and, if there is room, that element will push everything else as far away as it can go can.

FlexBox

One-dimensional layout
Either vertically or horizontally

- Lists
- Navigation
- Aligning elements along one axis
- Flexible responsiveness
- Micro layouts

Grid

Two-dimensional layout
Simultaneous control over horizontal and vertical directions

- General page structure
- Complex layouts
- Grid with strict size control of elements
- Repeating and template layouts

Grid container, grid cell and grid item

Grid container

Create a grid container by setting the display property with a value of grid or inline-grid. All direct children of grid containers become grid items

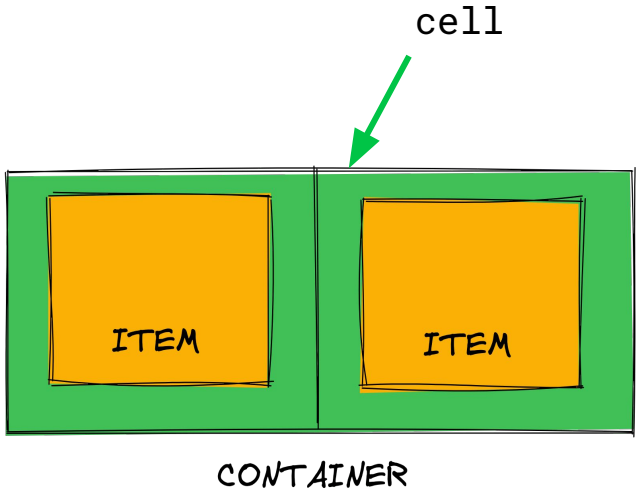
```
.container {
  display: grid;
  display: inline-grid;
}
```

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.

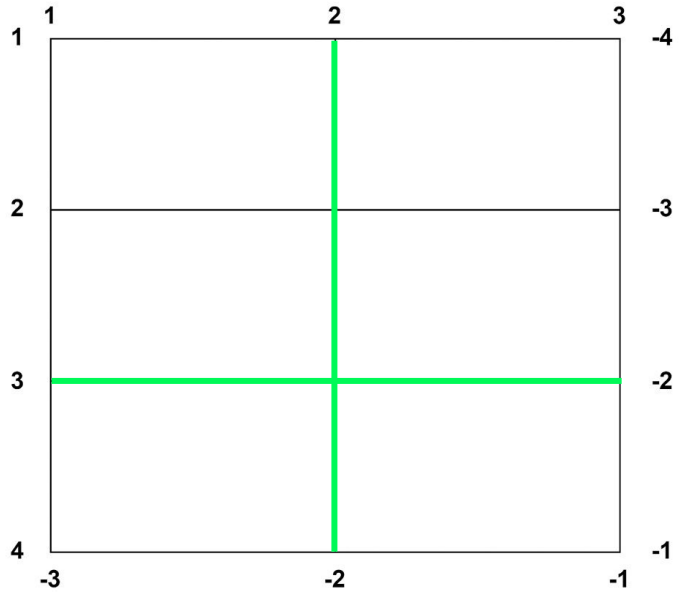
Grid item

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



Grid lines

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.



- Left to right / top to bottom – even numbers
- Right to left / bottom to top – negative numbers
- There are always one more line than the number of columns or rows
- These lines can be used to position elements within the grid.

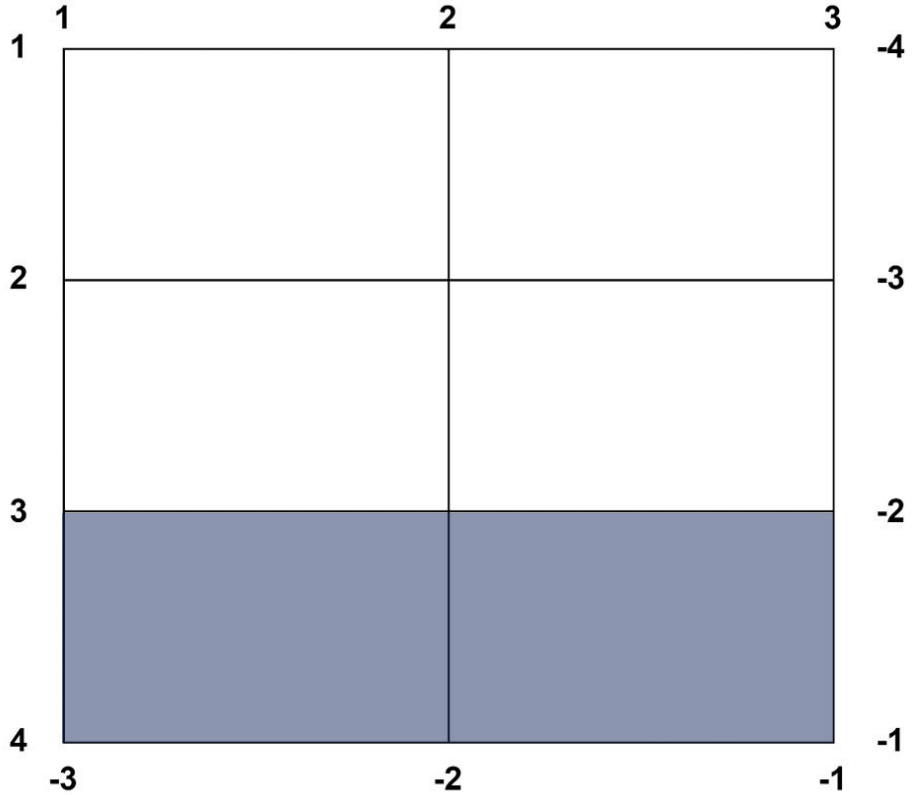
You can give custom names to lines using:
`grid-template-rows`
`grid-template-columns`

Grid track

Between two adjacent parallel grid lines, we see rows and columns of grid cells, also known as tracks. You can think of them as the columns or rows of the grid.

Either horizontally or vertically

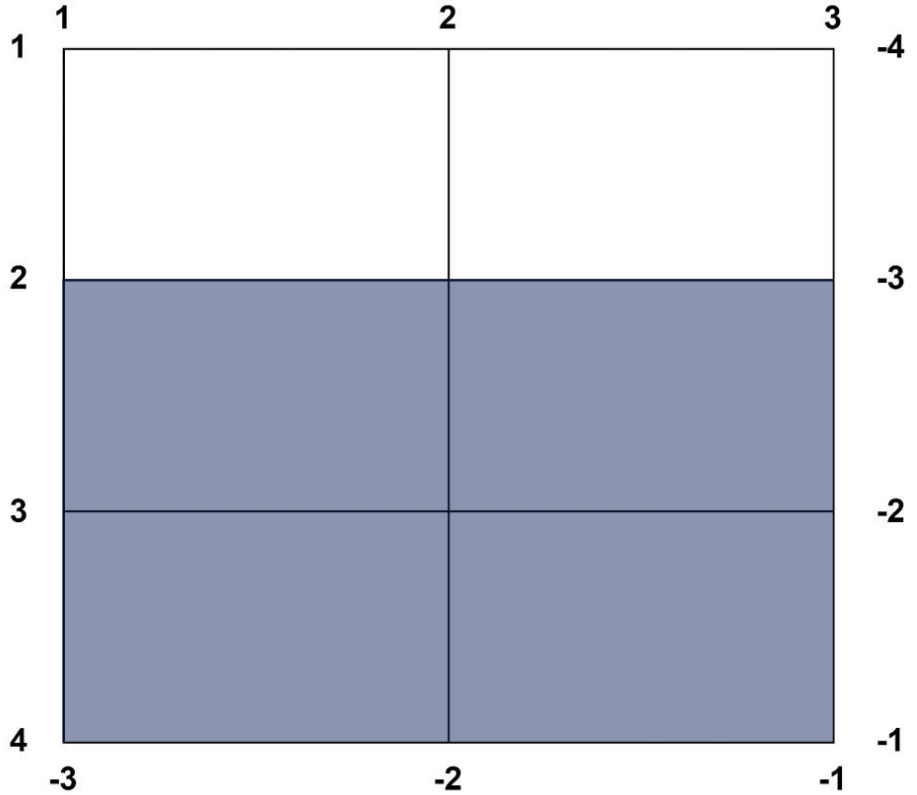
columns / rows



Grid area

The total space surrounded by **four grid lines.** A grid area may be composed of any number of grid cells.

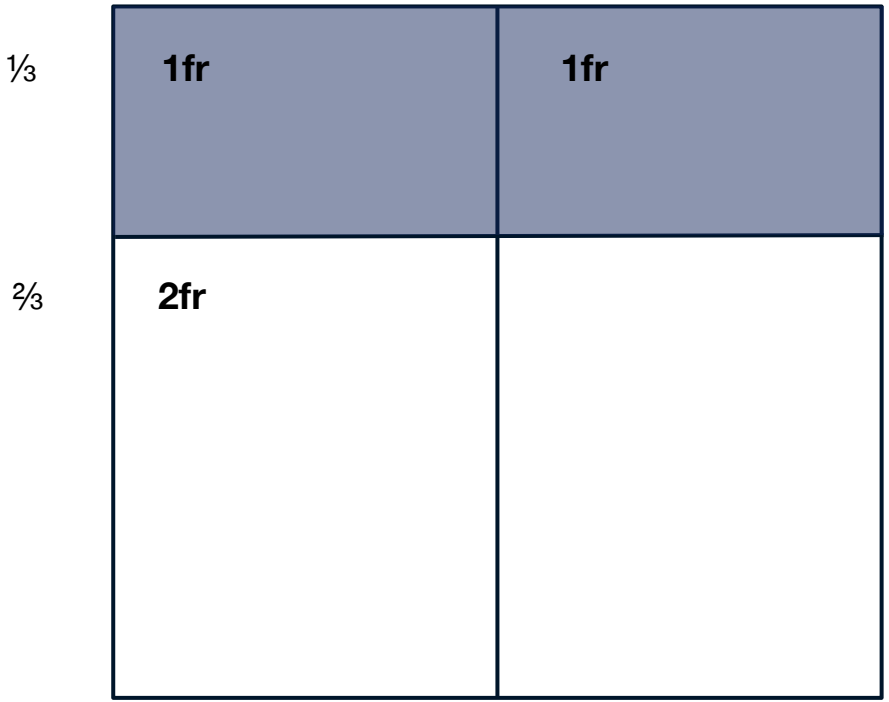
Can contain any number of cells.



Grid fr

The fr stands for 'fraction' and is a flexible unit that allows us to distribute the available free space within a grid container. Unlike fixed units like pixels or percentages, fr helps in creating layouts that adapt more dynamically to the container size.
The main difference between percentages and fractional units lies in the basis of their calculation. While percentages use the entire size of the grid to calculate their exact values, fr uses what remains after subtracting all fixed-size elements.

The fr unit only works within a grid context, making it a powerful tool for creating responsive and flexible layouts without needing to specify exact pixel values. This approach can be especially helpful when working with complex designs or when trying to maintain a consistent look across different screen sizes.

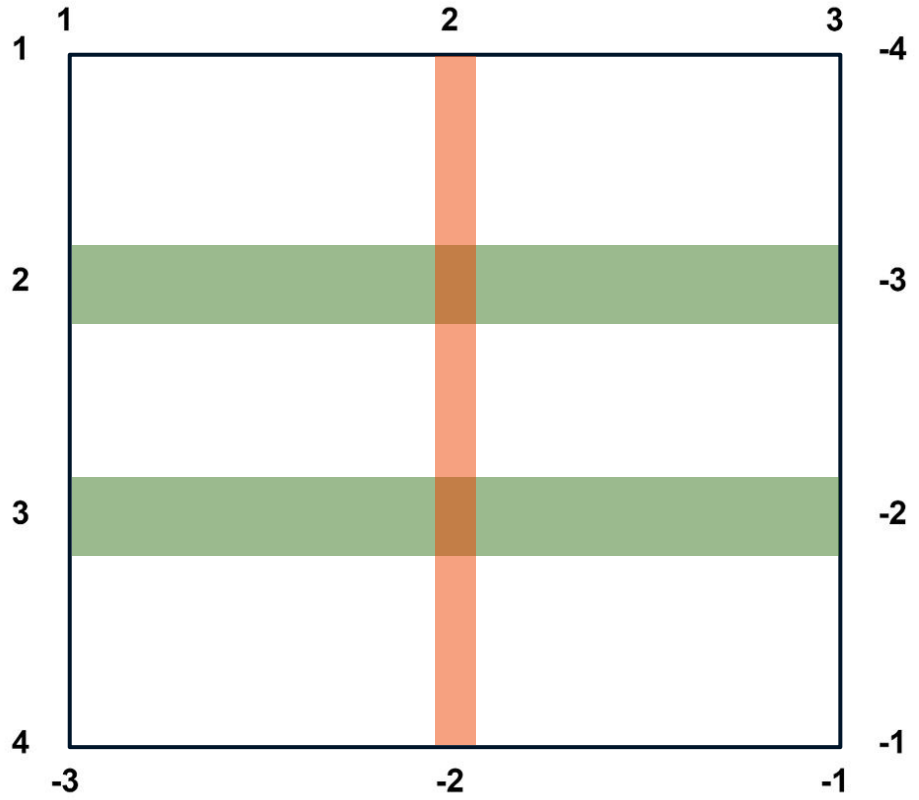


Grid gap

CSS Grid implements grid-gap which creates a gutter between your elements.
– Determines the distance between grid cells

Applied to a **grid container**

```
gap: 10px 20px;
gap-row: 10px;
gap-column: 20px;

px, rem, em
```



*

Grid gap (fun fact)

`gap - row limit (sum)`

Calculated by the formula:

This limit can either converge to a single value or diverge to infinity if s (gap) is greater than or equal to 100%.

Therefore, it's better to use **non-relative** units of measurement.

$$s = Gap\ size\ in\ percentage$$

$$\mathrm{g} = \mathrm{Sum\ of\ the\ gaps}$$

$$e = Width\ of\ elements\ without\ gap$$

$$g_0 = 0$$

$$g_n = s * (g_{n-1} + e)$$

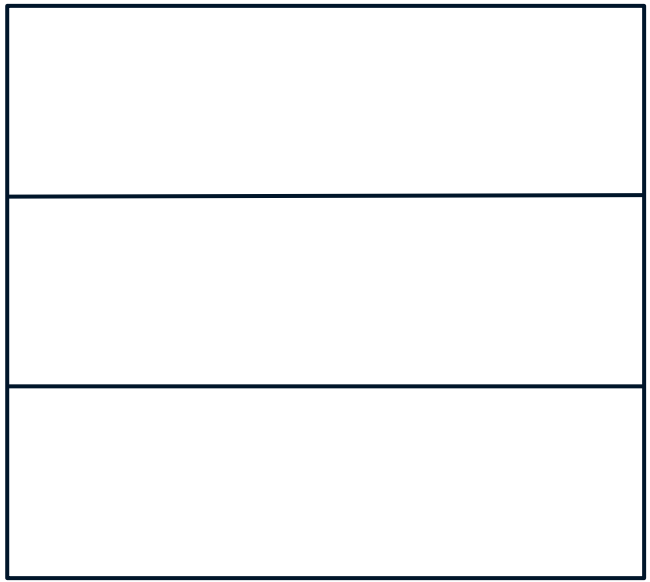
$$g = \lim_{n \to \infty} g_n$$

Creating a Grid

Grid layout

To create a grid layout, we apply display: grid; to the .grid-container. This property turns the container into a grid, and by default, each child element, or grid-cell, will be placed into its own grid cell. In this example, without any additional properties, our three grid-cell divs are automatically placed into three rows.

```
<section class="grid-container">
    <div class="grid-cell"></div>
    <div class="grid-cell"></div>
    <div class="grid-cell"></div>
</section>

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



Grid cell

In many ways, CSS Grid behaves like Flexbox. It requires a grid container and elements to align.
CSS Grid fully takes over the placement and sizing of statically and relatively positioned elements within the grid. This, in turn, means that any other inline positioning methods are no longer applicable. Any use of display:

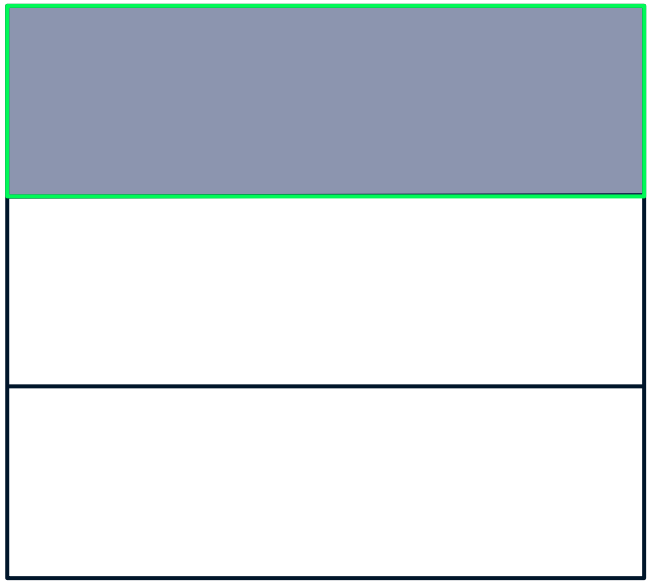
- **`display: inline-block`**
- **`display: table-cell`**
- **`float`**
- **`vertical-align`**

will have no effect.

However, we can still use

- position: fixed,
- position: absolute,
- or position: sticky,

as the elements are no longer static. Elements with position: relative can still be moved, and they are still affected by the grid



Grid columns/rows (base properties)

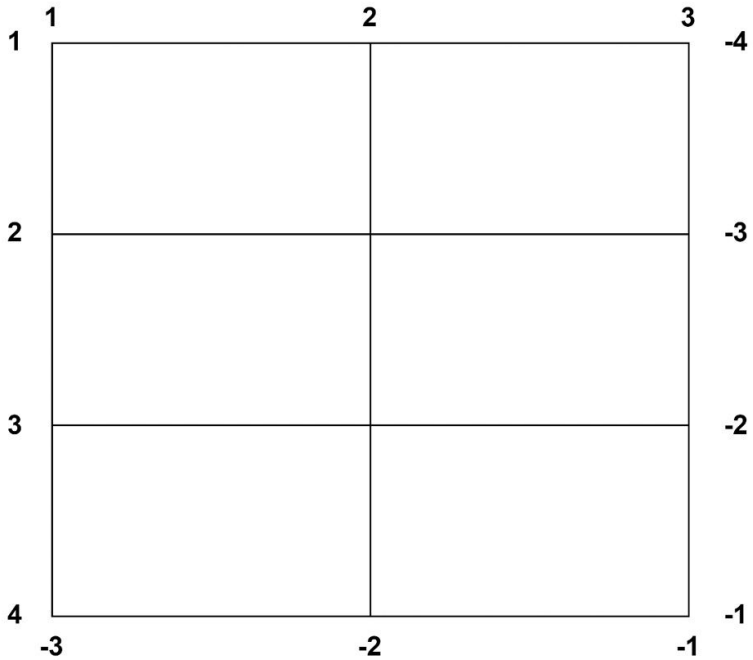
Grid systems include two key components: rows and columns. Rows are used to accommodate the columns. Columns make up the final structure and contain the actual content.

To be able to move elements, we first need to explicitly define the number of rows and columns in the grid. We achieve this by using the grid-template-rows and grid-template-columns rules, which go into the container element and pre-create the grid structure for us. We specify the number of grid columns and rows by defining their size, separated by spaces. The size can vary from row to row and column to column.

```
grid-template-rows: 1fr 1fr 1fr;
grid-template-rows: 200px auto 1fr;

grid-template-columns: 1fr 1fr;
grid-template-columns: 50% 20px auto;
```

Values in units:

- px
- %
- fr
- auto



Grid lines

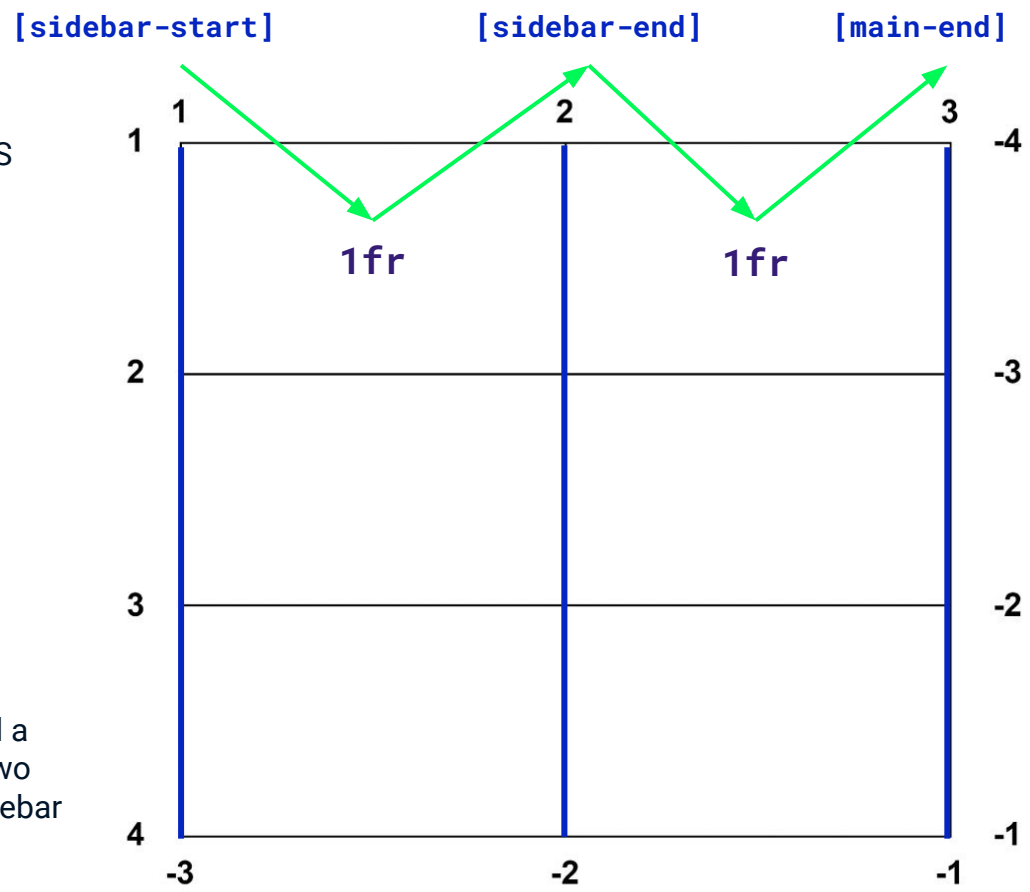
To add more clarity to our grid definitions, CSS Grid allows us to name columns and rows.

`grid-template-rows`
`grid-template-columns`

Naming goes from left to right:

```
grid-template-columns:
  [sidebar-start]
  1fr
  [sidebar-end]
  1fr
  [main-end];
```

anything inside square brackets is considered a name. We can also see that we've assigned two names to the middle grid line between the sidebar and the content, which we can separate with spaces.

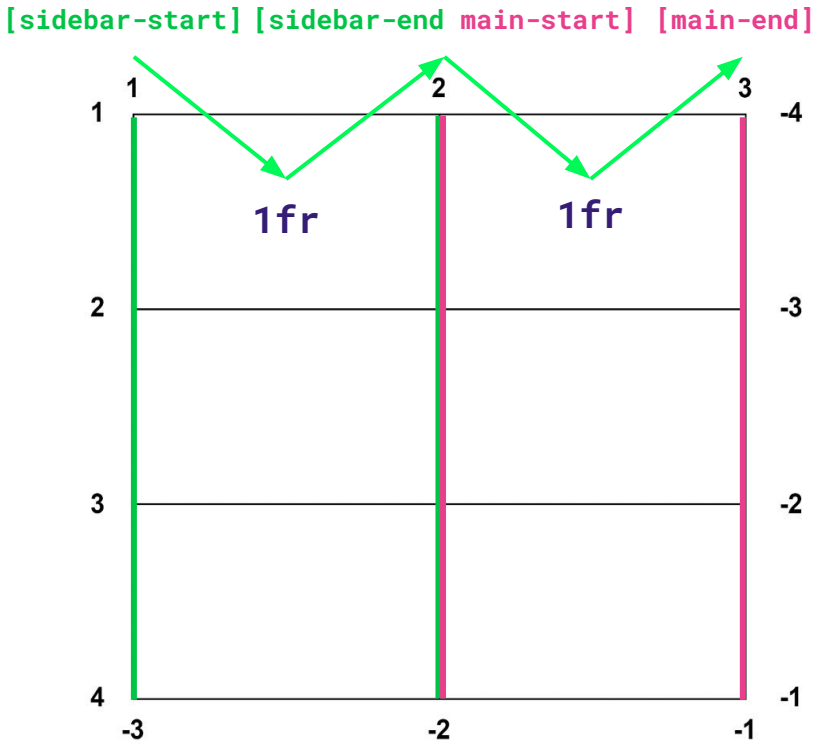


Grid lines with Two Names

A line can have two names:

```
grid-template-columns:
  [sidebar-start]
  1fr
  [sidebar-end main-start]
  1fr
  [main-end];

header {
    grid-column-start: sidebar-start;
    grid-column-end: main-end;
}
```

Code becomes more readable



Grid functions

Grid repeat

This function allows repeating a specific pattern of columns or rows. It reduces the need to write repetitive code.

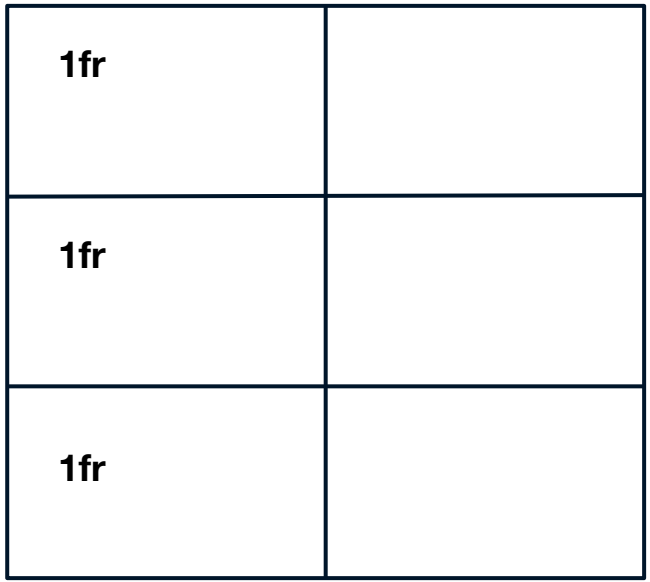
This grid definition has several issues. First, there's readability. The brain can quickly count only up to four items at a glance (this concept is known as subitizing), so when more than four or five similar items appear in an unfamiliar sequence, the brain doesn't immediately recognize the number of items and has to count them manually.

The second issue is maintainability. Repeating code is generally detrimental to maintainability, as it forces us to change the same thing multiple times.
The repeat() notation accepts 2 arguments: the first represents the number of times the defined tracks should repeat, and the second is the track definition.

```
grid-template-rows: repeat(3,1fr);
=
grid-template-rows: 1fr 1fr 1fr;
```

`repeat(3, 50px 100px)`
6 rows: **3** odd **50px, 3** even - **100px**

In the last crossed-out example, it shows an incorrect usage to emphasize that each repetition within repeat() applies to the full pattern, not individual components.
~~**`repeat(3 100px repeat(3, 1fr) 100px)`**~~



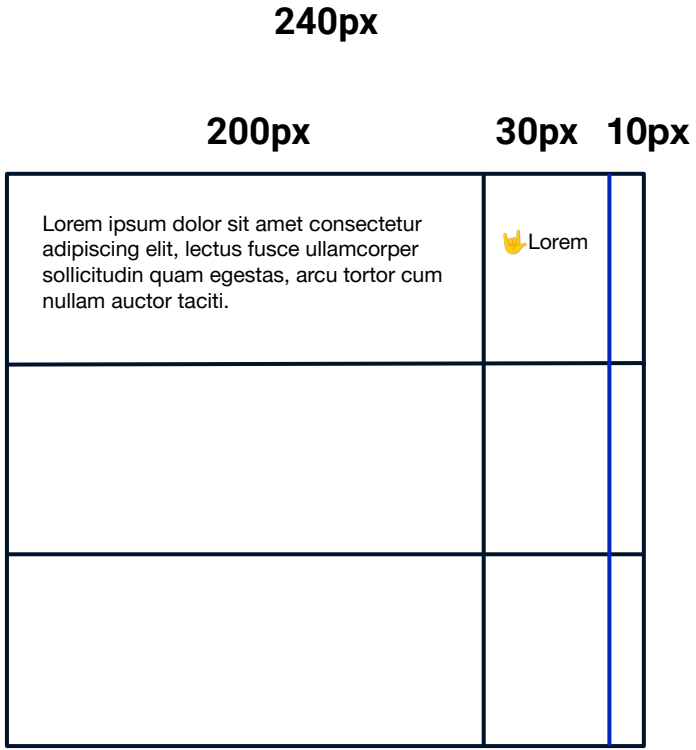
Grid fit-content()

This function allows you to set a size that is the 'best possible' within the given available space. The fit-content() function allows a grid cell to take up as much space as its content requires, but with a limit defined by a maximum size.

```
grid-template-columns: fit-content(200px);
```

This function is particularly useful when you want to create responsive designs where elements adjust to their content but still maintain control over the maximum size they can occupy. It's a balance between flexibility and structure, allowing your layout to adapt to varying content sizes while avoiding overflow or excessive stretching

fit-content is a mix of min-content and max-content. It behaves like the fr unit but with a minimum value of min-content and a maximum value of max-content.



Grid max-content

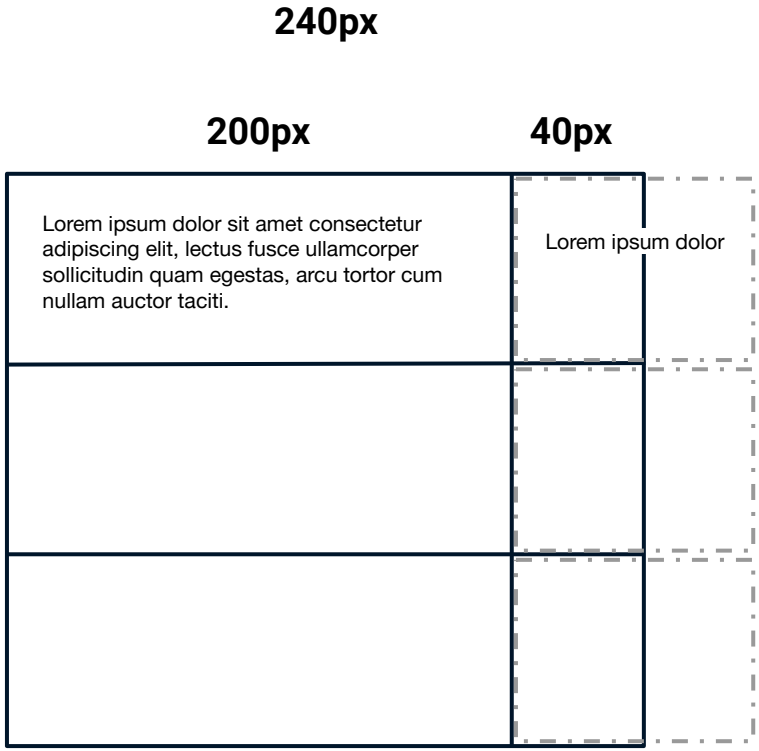
Now let's take a look at the max-content value in CSS Grid. The max-content property forces an element to take up as much space as it needs to display all of its content without any cutting or clipping.

```
grid-template-columns: fit-content(200px) max-content;
```

Thus, instead of using the width of the longest word, the width of the entire sentence will be calculated and used. In combination with images, it will use the maximum width or height of the image, depending on whether we are setting the size of a row or a column.

If the content is minimal, the cell may shrink, and if the content is extensive, the cell might overflow the container if it surpasses the available space. This makes max-content particularly useful when you want to ensure that no content is hidden, but it also requires careful consideration of the container's size and potential overflow issues.

In cases where content can be unpredictable in length, using max-content ensures that everything is visible, but you may need additional strategies, such as scrollbars or truncation, to handle excessive overflow.



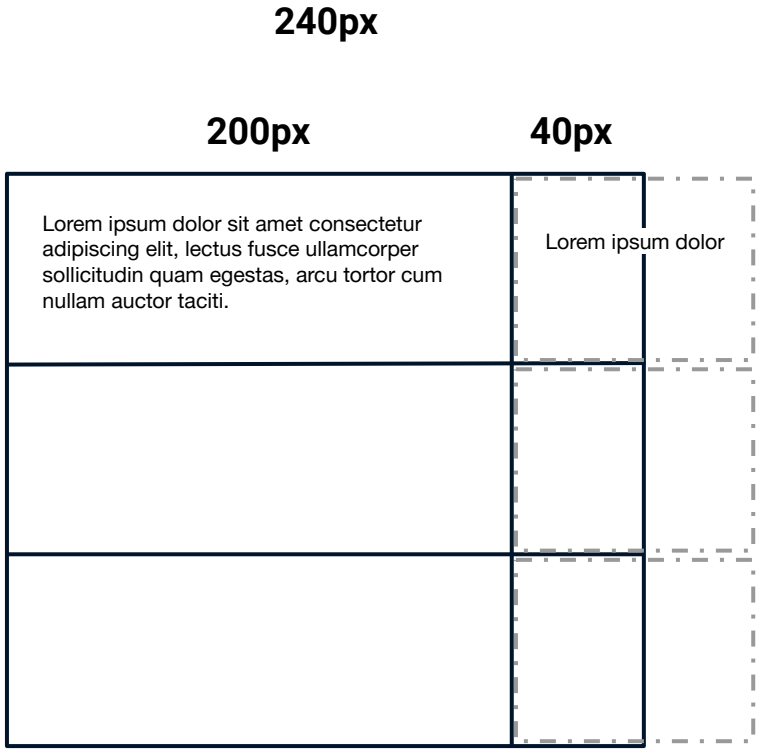
Grid min-content

The min-content property makes the content take up the minimum amount of space necessary. Specifically, it shrinks the column or row to be as small as possible while still allowing the longest word or unbreakable content to fit.

```
grid-template-columns: fit-content(200px)
max-content;
```

This behavior is particularly useful when you want to ensure that content does not take up more space than necessary, which is great for designs where you need to conserve space or avoid large gaps caused by short content. It also ensures that the layout remains flexible and adapts based on the content it contains.

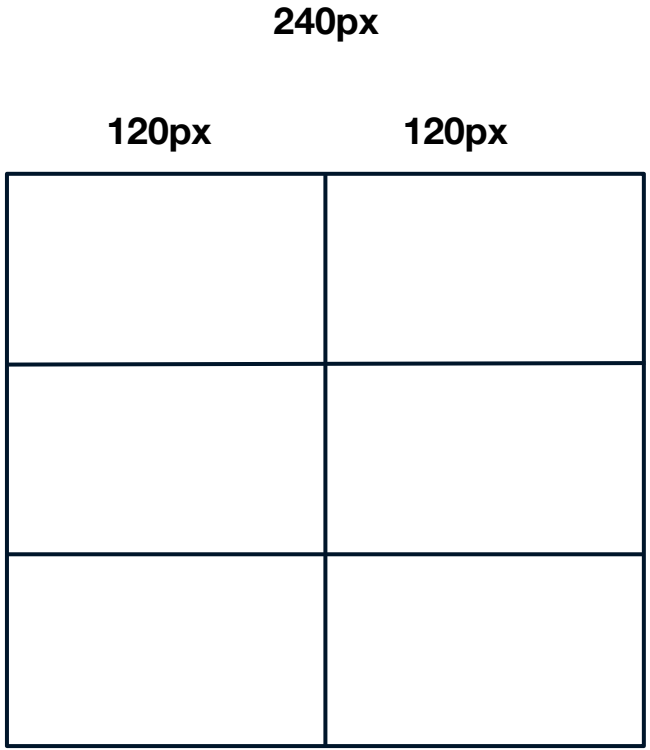
min-content also considers text. For example, if the minimum width of a grid cell is determined by the longest word in that cell, the pixel width of that word will then be the minimum size. This can have interesting effects when combined with word-break: break-all; since the minimum width is then calculated based on the widest character



Grid auto

The auto value sets the size of rows or columns based on the content within them. If there's a lot of content, the cell will take up as much space as necessary, potentially filling the entire available space.

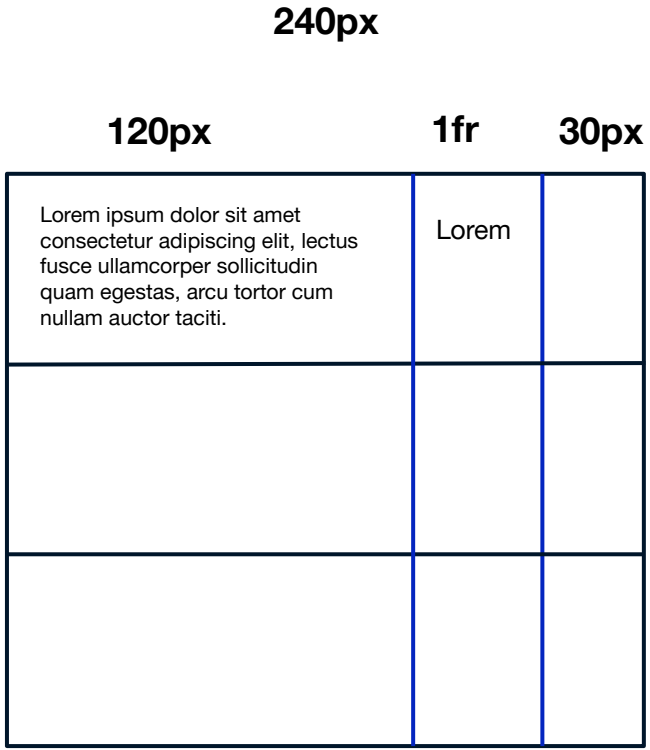
```
grid-template-columns: auto auto;
=
grid-template-columns: 1fr 1fr;
```



Grid auto

– auto
The cell stretches to the full available width/height of the container

```
grid-template-columns: auto 1fr 30px;
```

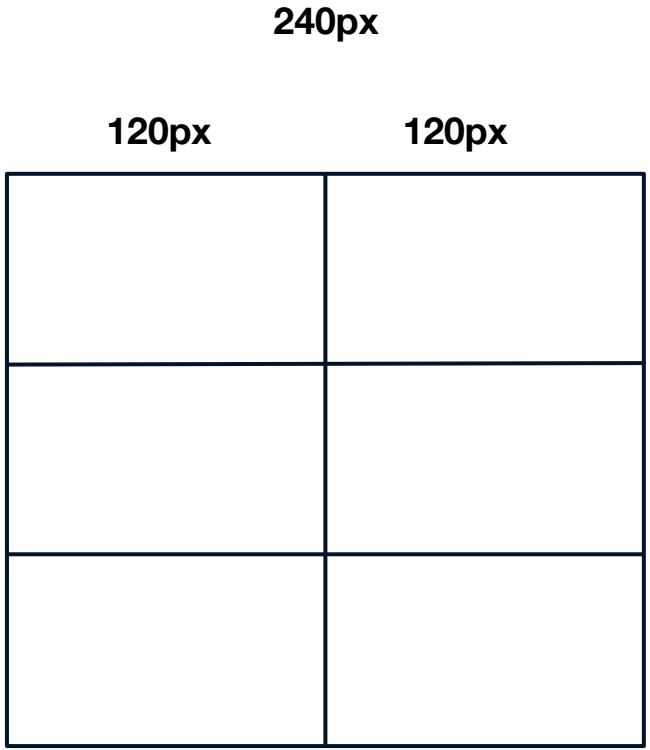


Grid minmax()

This function is used to set the minimum and maximum size for columns or rows. It is useful for creating responsive layouts that look good on screens of different sizes.
In combination with the fr unit, we can set cells to have relative sizes while maintaining minimum and maximum sizes for them.
The minmax() function accepts 2 arguments: the first is the minimum size of the track and the second the maximum size.
Alongside length values, the values can also be auto, which allows the track to grow/stretch based on the size of the content.

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

This approach allows us to maintain control over the minimum size, ensuring that elements don't become too small, while also allowing for flexibility and responsiveness as the layout adjusts to different screen sizes or container widths.

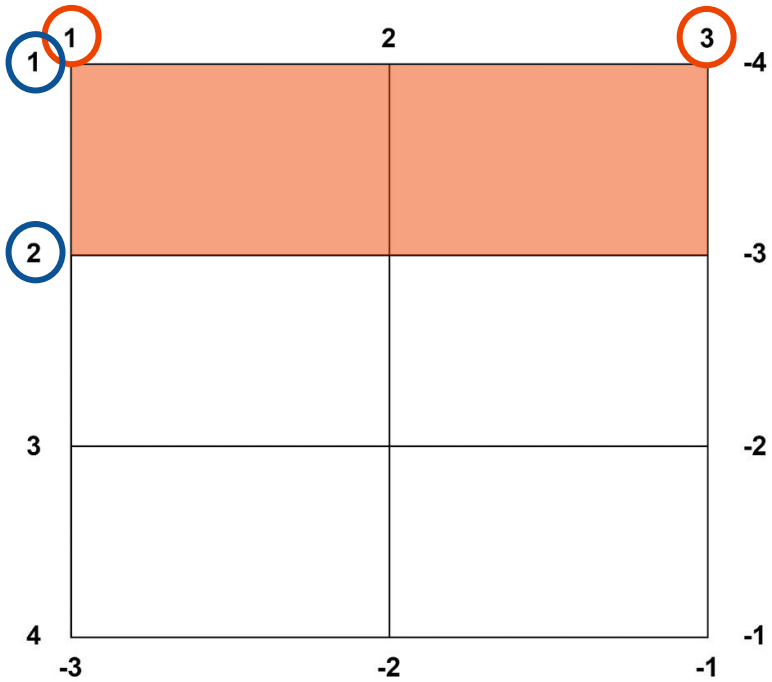


Positioning Elements in the Grid

Grid positioning

– how to manage the size and placement of a grid cell by specifying the lines between which the element spans.
We use the properties grid-column-start, grid-column-end, grid-row-start, and grid-row-end to define the exact lines that the grid item will start and end on. This is particularly useful when you need specific control over the layout, such as placing a header across multiple columns.
Here, the element spans from column line 1 to 3, and from row line 1 to 2. This allows you to precisely control where elements are placed within the grid.

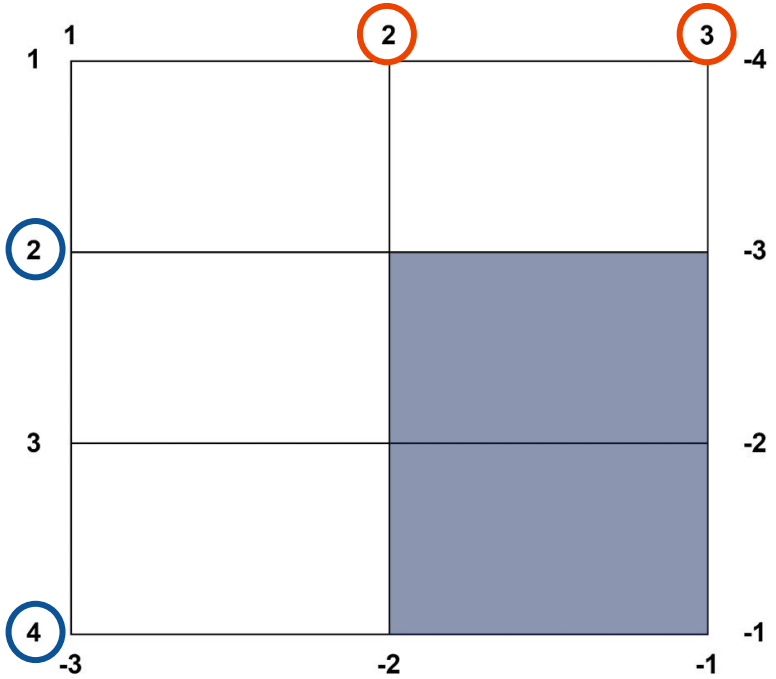
```
grid-column-start: 1;
grid-column-end: 3;

grid-row-start: 1;
grid-row-end: 2;
```



Grid positioning

Shortened notation:

```
grid-column: 2 / 3;

grid-row: 2 / 4;
```



Here, we explore different types of positioning values. You can use:

- **Line number**

```
grid-column: 1 / 3;
```

- **span - the number of occupied cells**

```
grid-column: 1 / span 2;
```

- **line-name**

```
grid-column: start / end;
grid-column: [start] / [middle];
```

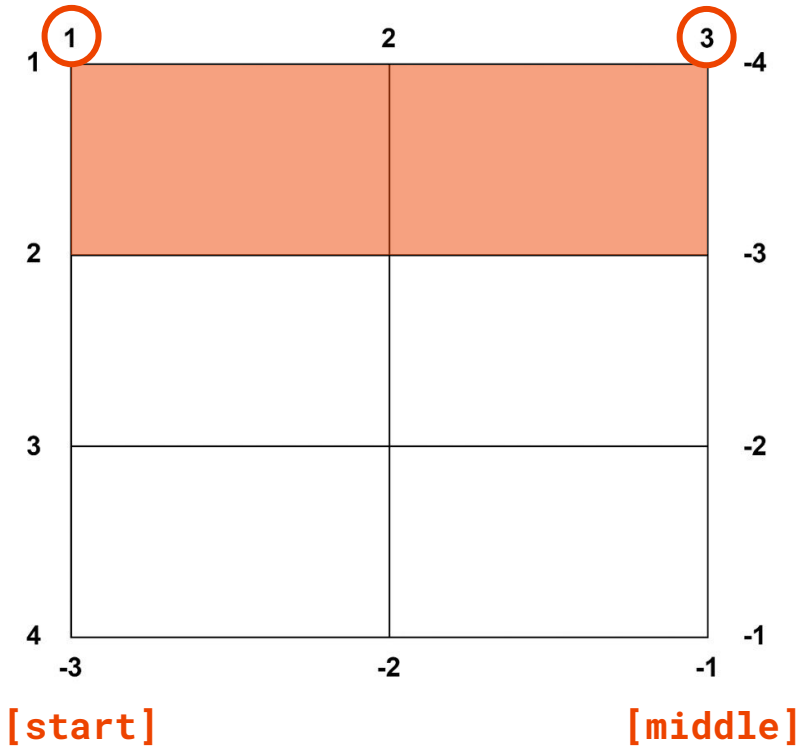
- **auto**

```
grid-column: 1 / auto;
```

It will occupy one cell, starting from the first line.

– If an element's position is not defined in the grid, it will be placed in the first available cell.

– The element will automatically determine where to start and end.



Grid-area

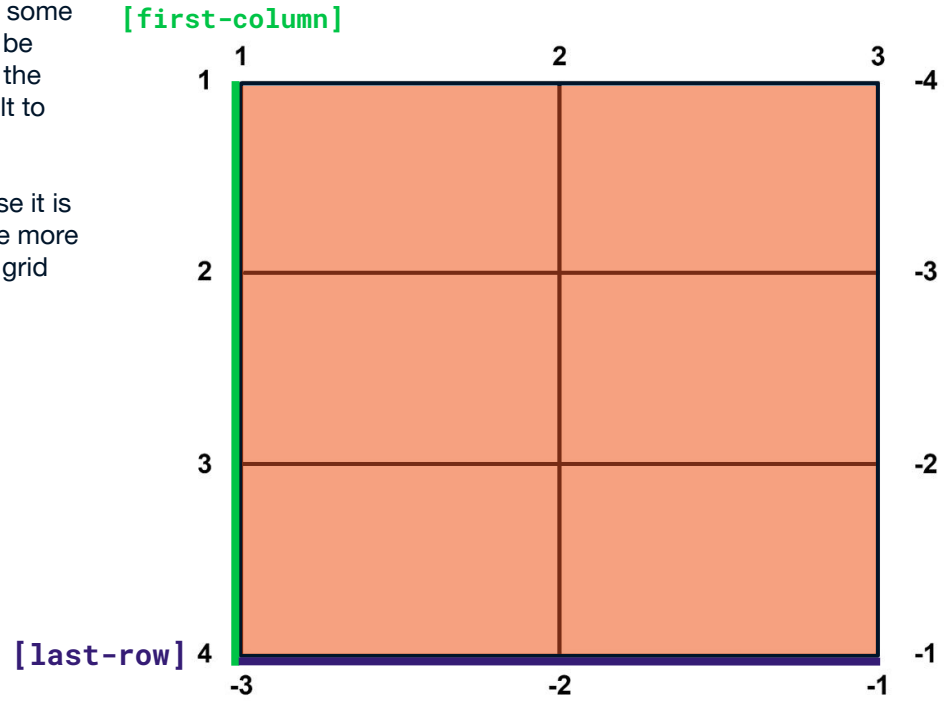
Working with larger grids can become confusing. So far, we have used two rules for the container to define the size of our grid, and then some seemingly arbitrary numbers to specify where an element should be placed in both rows and columns. For developers unfamiliar with the code, these seemingly arbitrary numbers and sizes can be difficult to understand.

They will need to spend more time understanding the grid because it is not very informative. However, CSS Grid offers us the ability to be more explicit. Now we will see how to name rows, columns, and entire grid areas.

`grid-area` **Shorthand property for:**
`grid-row-start +`
`grid-column-start +`
`grid-row-end +`
`grid-column-end`

```
grid-area: row-start / column-start / row-end /
column-end;
```

```
grid-area: 1 / first-column / last-row / 3;
```



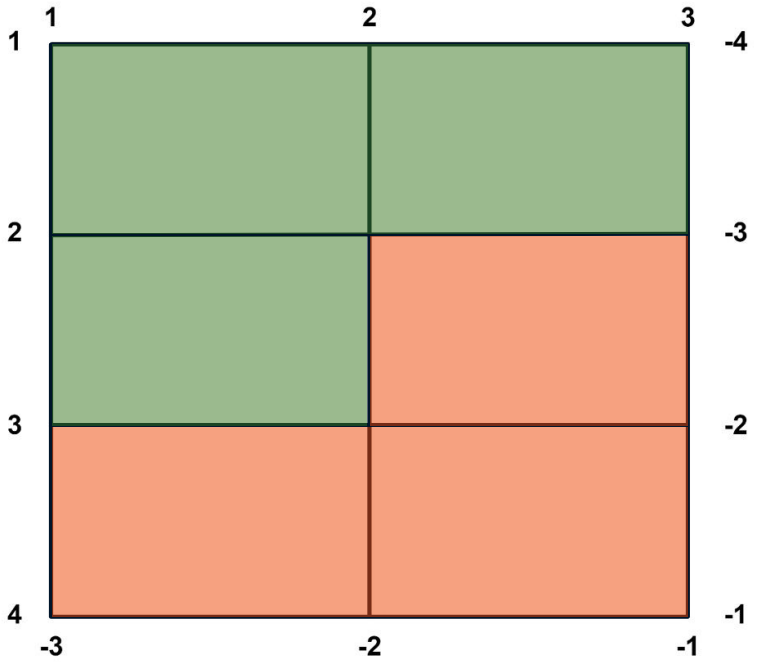
Grouping Elements in the Grid

display: contents

This property instructs the browser to ignore the element itself and instead use its child elements as grid items. This is particularly useful when you want to group grid elements or apply different styles to these groups without affecting the grid structure.

The display: contents property is applied to a direct descendant of the grid container. It allows us to create logical groupings within the grid without adding unnecessary wrapper elements that might interfere with the grid layout.

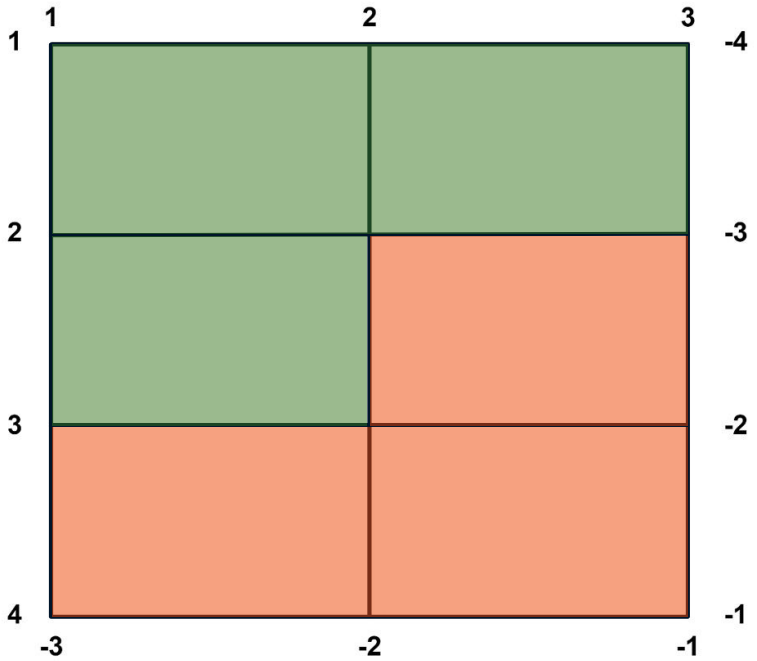
In the example shown, the areas highlighted in green and orange represent grouped elements within the grid. Despite the grouping, each child element still behaves as an individual grid item, taking up its own space in the grid. This approach is beneficial when you need to manage the styling or behavior of grouped elements without disrupting the grid's flow.



display: contents

```
<div class="grid-container">
    <section class="new-articles">
        <article></article> x3
    </section>
    <section class="old-articles">
        <article></article> x10
    </section>
</div>
```

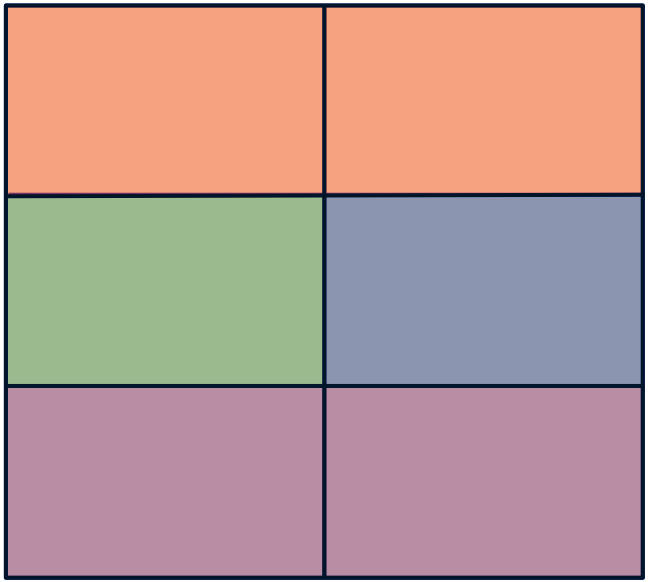
```
section {
    display: contents;
}
```



Grid areas

Grid template

This property instructs the browser to ignore the element itself and instead use its child elements as grid items. This is particularly useful when you want to group grid elements or apply different styles to these groups without affecting the grid structure.
The display: contents property is applied to a direct descendant of the grid container. It allows us to create logical groupings within the grid without adding unnecessary wrapper elements that might interfere with the grid layout.
In the example shown, the areas highlighted in green and orange represent grouped elements within the grid. Despite the grouping, each child element still behaves as an individual grid item, taking up its own space in the grid. This approach is beneficial when you need to manage the styling or behavior of grouped elements without disrupting the grid's flow.



`grid-template-areas` define the grid visually

applied to **grid container**

```
grid-template-areas:
“header header”
“sidebar content”
“footer footer”;
```

Grid area

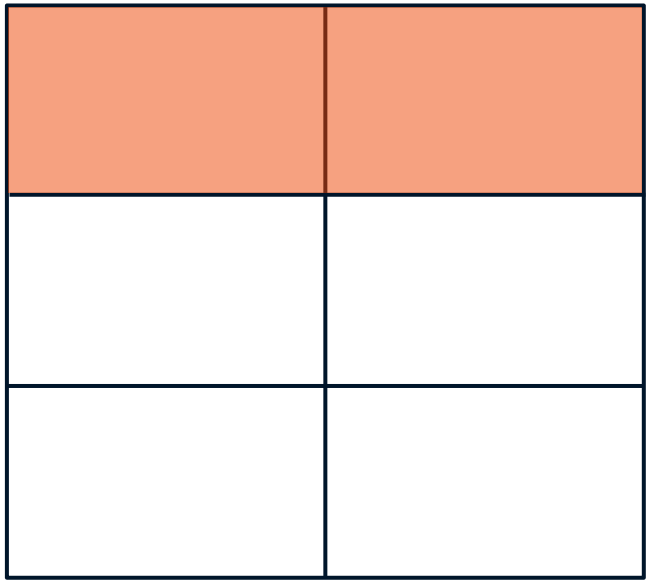
The area name is set for the grid cell

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

It's better to name based on the function the cell performs

Bad Names:
a
item1
block-c

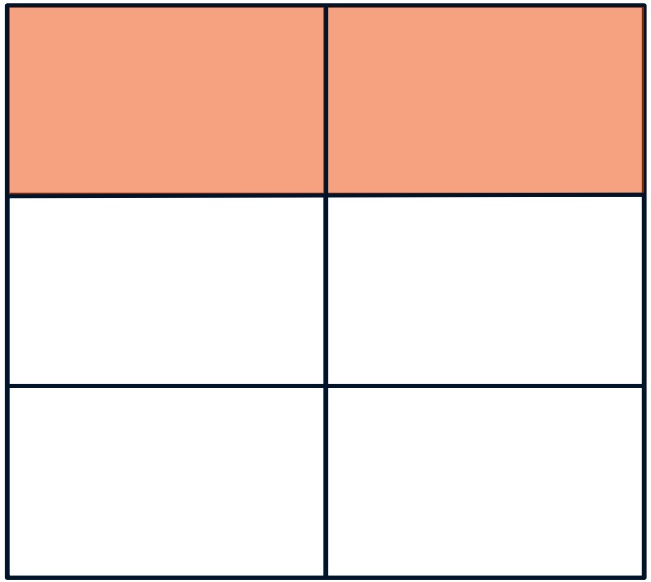
Good Names:
catalog-item
article-header
footer-logo



Empty Grid Cells

We can ignore a cell or leave it as an empty cell using the full-stop character (.)
If that feels visually awkward or imbalanced to you, we can use multiple full-stop characters without any whitespaces separating them.
A named cell token can span multiple grid cells, But those cells must form a rectangular layout. In other words, we're unable to create "L" or "T"-shaped layouts, although the spec does hint at support for non-rectangular layouts with disconnected regions in the future.

```
grid-template-areas:
"header header"
"sidebar .";

can use multiple full-stop:
"sidebar ...";

don't use just space:
"sidebar ";
```

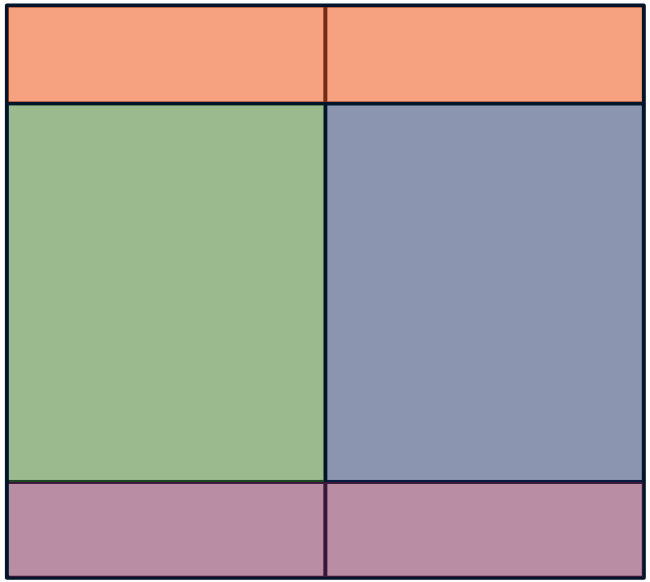


Grid areas shortland

CSS Grid provides shorthand properties that combine multiple other properties into one. This simplifies writing and reading code.

grid-template
= grid-template-areas +
grid-template-rows / grid-template-columns;

```
grid-template:
“header header” 100px
“sidebar content” 1fr
“footer footer” 100px / 50% 1fr;
```



Implicit grids

Implicit grids

When working with CSS Grid, the first thing to do is to set display: grid on the element that we want to be become a grid container. Then we explicitly define the grid using a combination of grid-template-columns, grid-template-rows, and grid-template-areas. And from there, the next step is to place items inside the grid.
This is the classic approach that should be used and I also recommend it. However, there is another approach for creating grids without any explicit definition. We call this the implicit grid.

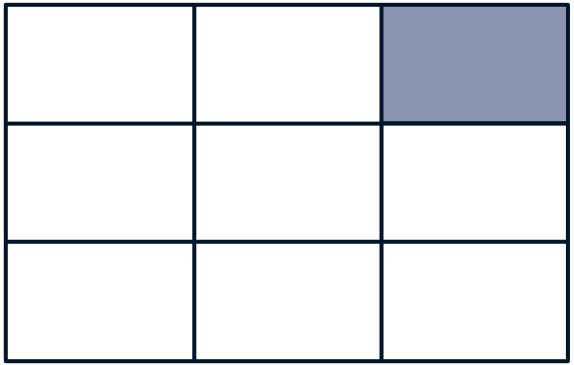
```
grid-template-columns / grid-template-rows
```

```
grid-template-columns: repeat(2, 1fr);
```

“Explicit, implicit? What the heck is going on here?”
The grid-template-rows, grid-template-columns, and grid-template-areas properties define a fixed number of tracks that form the explicit grid. When grid items are positioned outside of these bounds, the grid container generates implicit grid tracks by adding implicit grid lines to the grid. These lines together with the explicit grid form the implicit grid.

So, in plain English, the browser auto-generates extra rows and columns in case any elements happen to be placed outside the defined grid.
When grid items are positioned outside of these bounds, the grid container generates implicit grid tracks by adding implicit grid lines to the grid.

If you have 7 child elements, 6 of them will fill the first 2 columns. For the 7th element, Grid will automatically create a 3rd column. This 3rd column is part of the implicit grid.



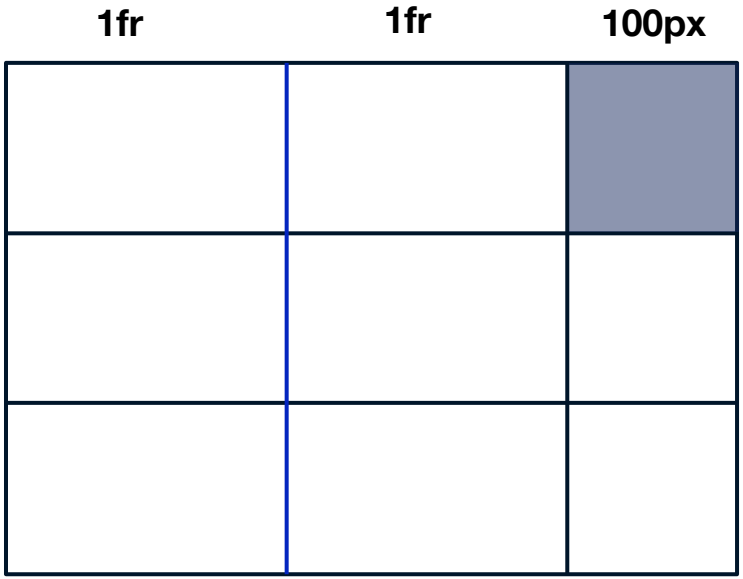
Managing Implicit Grids

The grid-auto-columns and grid-auto-rows properties are used to control the sizes of automatically created columns and rows.

Elements are automatically placed in the implicit grid based on the order they appear in the source code. However, by using the grid-column and grid-row properties, you can control exactly where the elements will be placed, including in the cells of the implicit grid.

grid-auto-columns / grid-auto-rows

```
grid-template-columns: repeat(2, 1fr);
grid-auto-columns: 100px;
```



Auto Placement in the Grid

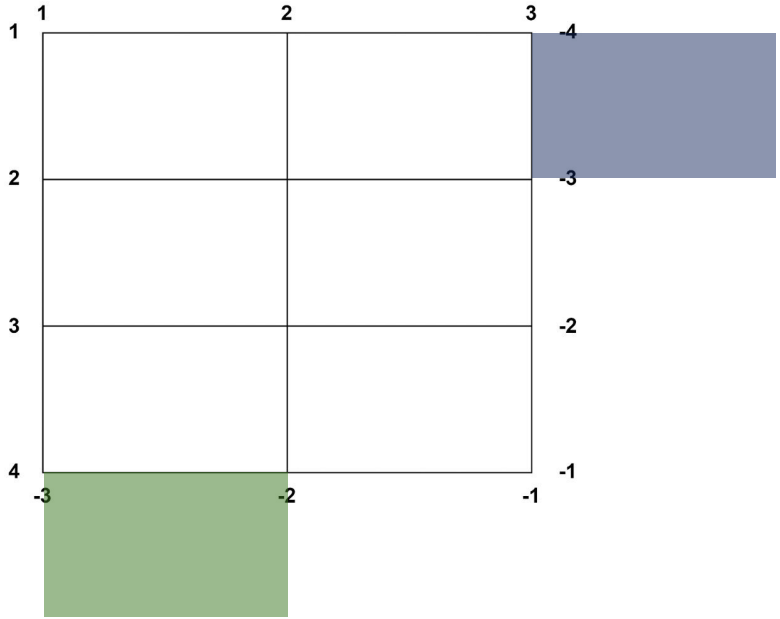
Grid auto flow

Similar to the concept of implicit grid, auto-placement is the ability of the browser to automatically place the items inside the grid. We don't always need to give the position of each item.
When grid items are positioned outside of these bounds, the grid container generates implicit grid tracks by adding implicit grid lines to the grid.

Until now, we've seen how CSS Grid places elements according to the same rule: it starts from the top-left corner and fills the grid with elements, row by row, until reaching the bottom-right corner. However, there is a way to control how CSS Grid places new elements in the grid, which is called the grid auto-flow.
This property controls how elements that have not been explicitly placed are positioned in the grid. The values can be row or column, and they determine whether a new row or a new column is filled when elements are automatically placed.

applied to **grid container**

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



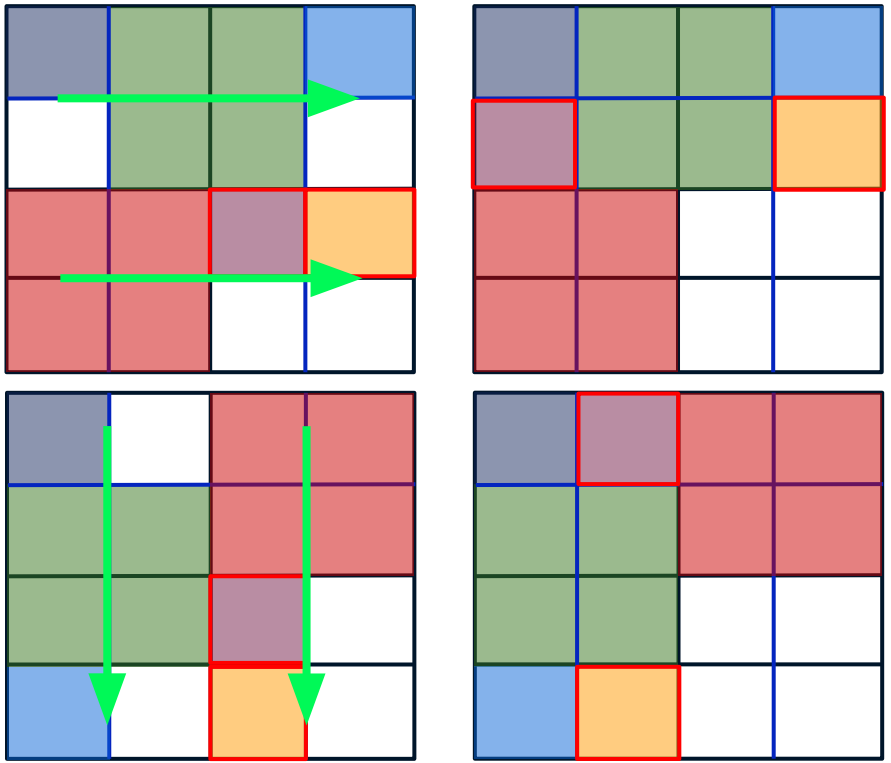
Grid auto flow

Dense attempts to fill any gaps in the grid, regardless of their location. This is useful if we have many elements of different sizes that don't need a specific order, such as randomized mood images on restaurant websites.
row dense and column dense:
These values are combinations of the previous. First, they will attempt to fill rows and columns respectively, and then fill any resulting gaps. Since row is the default value, row dense is equivalent to simply using dense.

The arrangement is similar: some parts are rotated and mirrored, while the rest are filled column by column. By thinking about this, we can work quite cleverly with grids. Perhaps there is some pixel art created using CSS Grid—who knows?
Grid flow gives us even more control over the grid's behavior when we don't know its content in advance. Think about user-generated content or randomly selected elements. Grid Flow fills any gaps for us and ensures a beautiful grid.

`grid-auto-flow: dense;` – tries to fill any gaps in the grid, regardless of their location.

`row dense;`
`column dense;` – they first attempt to fill rows and columns, respectively, and then fill any remaining gaps.



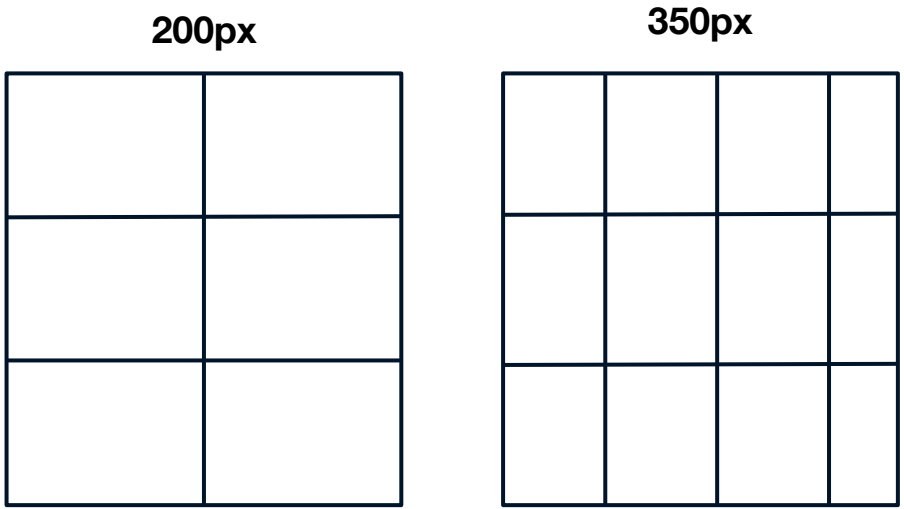
Grid auto-fill

The auto-fill property is used with grid-template-columns or grid-template-rows to automatically create columns or rows based on the available space.
The auto-fill value will add as many grid cells as possible into a row or column.

used with:
`grid-template-columns`
`grid-template-rows`

```
grid-template-columns: repeat(auto-fill, 100px);
```

This feature is particularly useful when you want to ensure that your layout is responsive and adapts to different screen sizes by automatically filling in as many columns or rows as possible without overextending.



Grid auto-fit

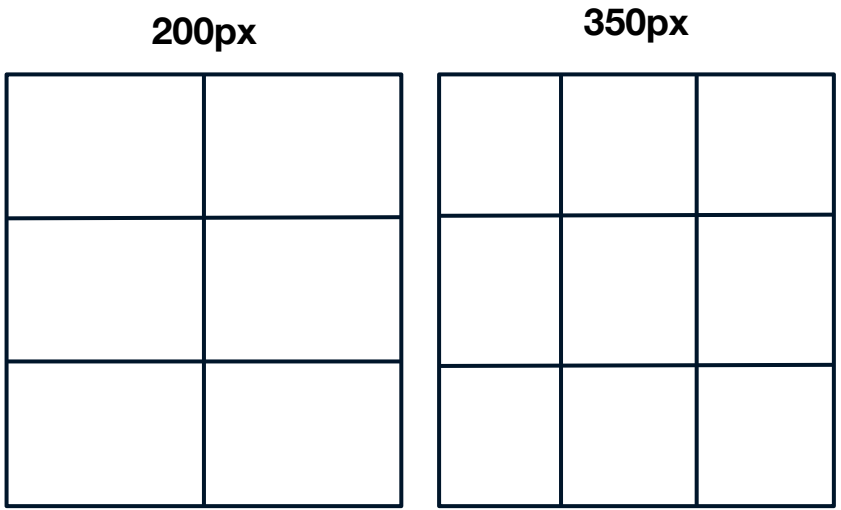
Unlike auto-fill, which fills the container with as many columns or rows as possible while leaving any extra space empty, auto-fit goes a step further by removing these empty columns or rows and redistributing the available space among the remaining elements.

On the other hand, we have auto-fit. This keyword tries to fill the entire width of the grid (when used with grid-template-columns) or height (when used with grid-template-rows). This value is usually convenient when working with minmax() and fr as size definitions, as it allows grid cells to expand and contract.

- auto-fill is best suited when you want to see empty cells in your design, such as if you have a tiled interface and want the tiles to always occupy the full width of the container.
- auto-fit is better when you want your design to look as full as possible, without empty cells.

used with:
`grid-template-columns`
`grid-template-rows`

```
grid-template-columns: repeat(auto-fit, minmax(100px, 1fr));
```



Aligning Elements Inside Cells

Grid align

Imagine that a grid is like a chessboard, where we have cells, and we want to place pieces (elements) in them. The question is how we want to align these pieces within their cells and how to align groups of pieces together.
You can control the alignment of cells and rows using almost the same properties as in Flexbox.
The key difference here is that in Grid, the axes are always fixed.

- The align-* properties align items along the column axis,
- while the justify-* properties align them along the row axis.

This consistent axis orientation helps in maintaining a predictable layout structure across different screen sizes and content arrangements.

- align-items and justify-items, which control the alignment of all grid items within their grid area.
- align-self and justify-self allow individual items to be aligned within their respective grid areas.
- align-content and justify-content align the entire grid's content within the grid container.

align-items + justify-items
align-self + justify-self
align-self + justify-self
align-content + justify-content

Grid align values

`start –` elements are aligned to the start of the container.

`end –` elements are aligned to the end of the container.

`center –` elements are centered.

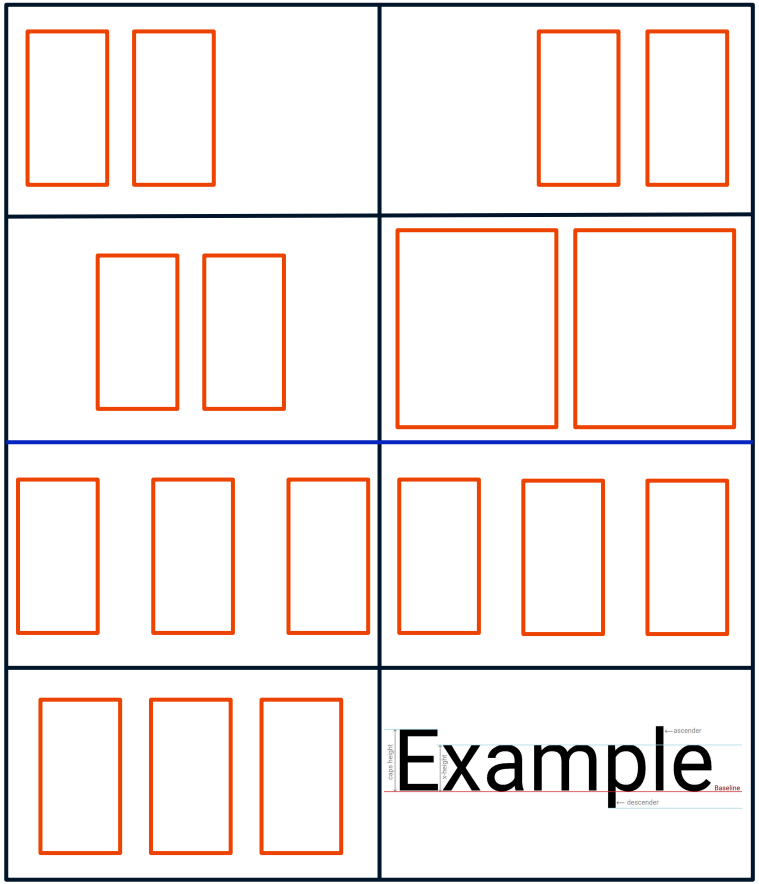
`stretch –` elements stretch to fill the entire available space of the grid item; this is the default value.

`space-between –` elements are distributed with equal spacing between them, and the outer elements are aligned with the edges.

`space-around –` elements are distributed with equal spacing between them, but the spacing at the edges is half of the spacing between the elements.

`space-evenly –` elements are distributed with equal spacing between them, with the same spacing at the edges as between the elements.

`baseline –` aligns grid items along the text's baseline in the entire row. This value is particularly useful for text of different sizes that needs to be aligned on the same line.



Grid place-items

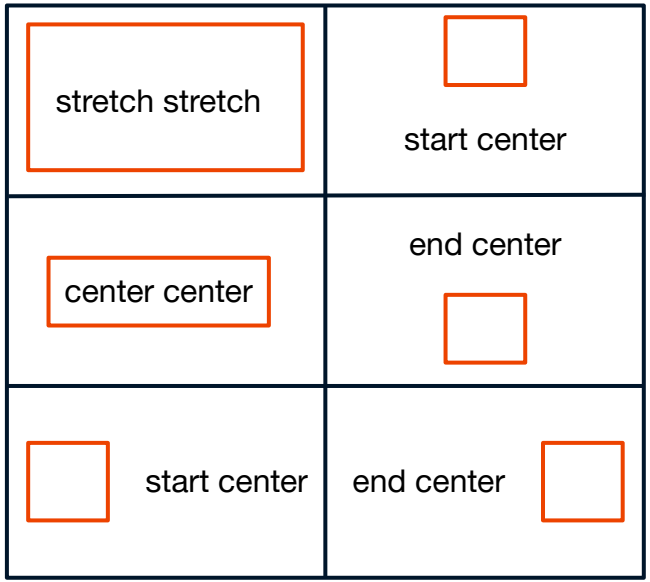
This is a universal property for aligning all elements within their grid cells.
Let's say you have a grid, and you want all the elements inside to be centered within their cells.
Instead of writing two separate properties align-items and justify-items, you simply write:

```
place-items: center center;
```

align-items + justify-items

applied to **grid container**

start
end
center
stretch



Grid place-self

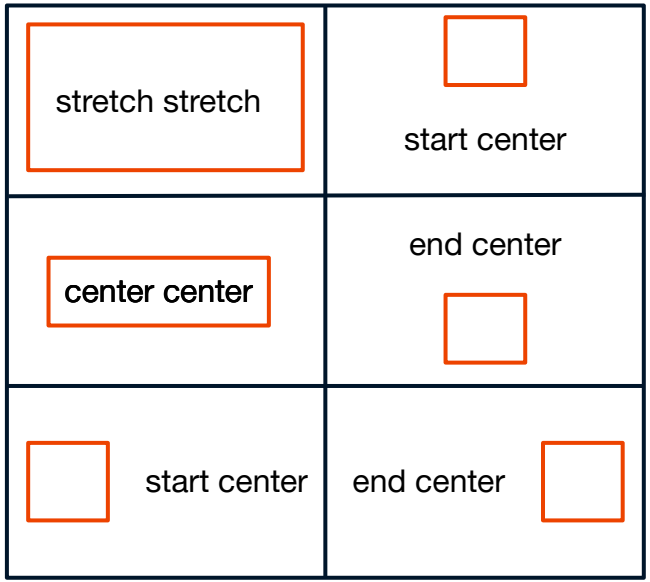
Now, what if we have one specific element, and we want to align it differently from the rest? Instead of writing align-self and justify-self, we can use place-self.
For example, if we want one element to be aligned at the start of its cell vertically but centered horizontally, we would write:

```
place-self: center center;
```

align-self + justify-self

applied to **grid cell**

start
end
center
stretch



Grid place-content

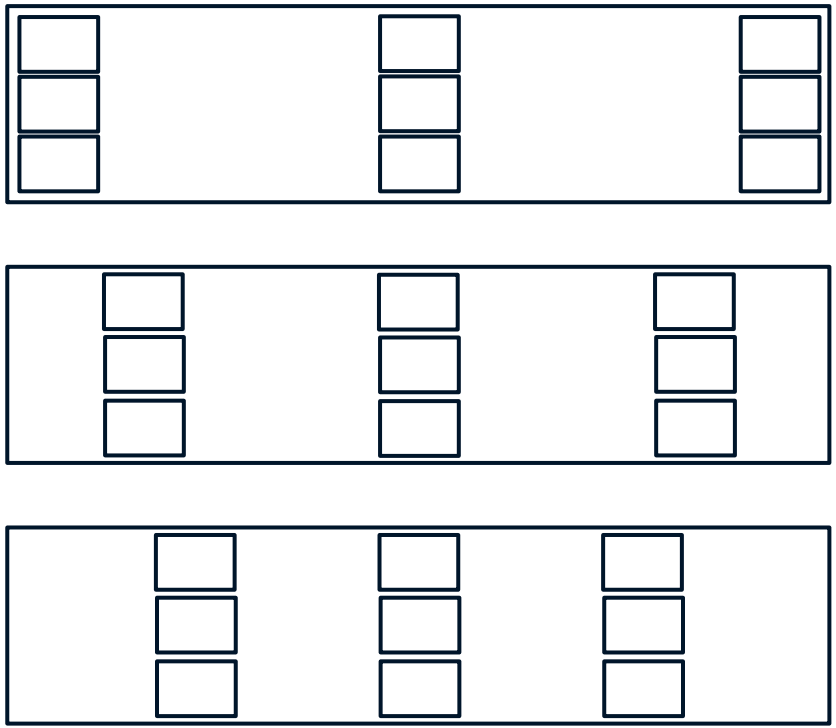
And finally, when we want to control the alignment of all elements in the grid as a group (instead of individual cells), we use place-content. This is a shorthand property for align-content and justify-content.
Imagine you have several rows and columns, and you want them to be evenly distributed across the grid. You simply write:

```
place-content: space-between center;
```

align-content + justify-content

applied to **grid container**

```
start
end
center
stretch
space-between
space-around
space-evenly
```



Grid shortland

Grid

```
grid =
grid-auto-flow +
grid-auto-rows +
grid-auto-columns +
grid-template-areas +
grid-template-columns +
grid-template-rows
```

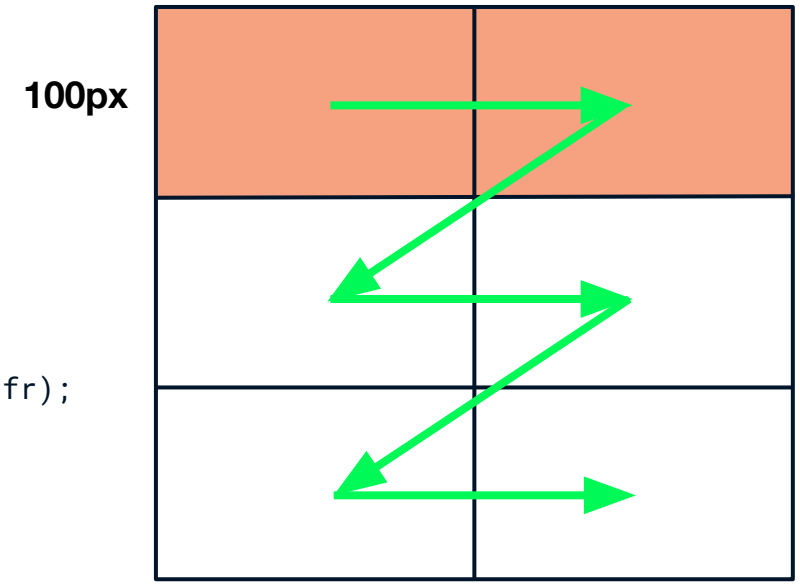
Shorthand properties help us reduce the verbosity of our code. They can help us avoid repetition and present our grid more clearly. Writing some of these shorthands, especially grid, may require more effort, but they can save us a lot of time once we get familiar with them. Their use is also optional—some developers prefer shorthands, while others do not.

\+ Less code means faster loading.

\- Shortened syntax is harder to read and understand.

Grid

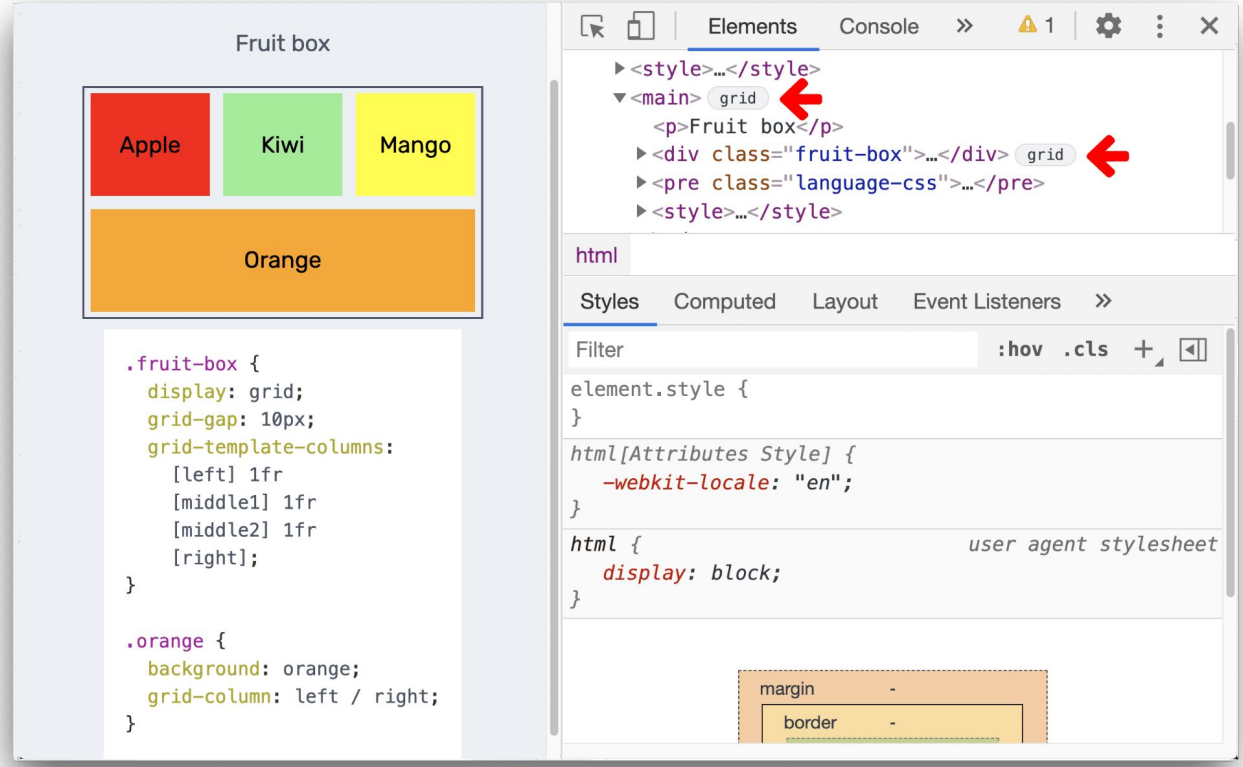
```
.grid {
      display: grid;
      grid-auto-rows: 100px;
      grid-template-columns: repeat(2, 1fr);
      grid-auto-flow: row dense;
}

.grid {
      display: grid;
      grid: auto-flow dense 100px / repeat(2, 1fr);
}
```



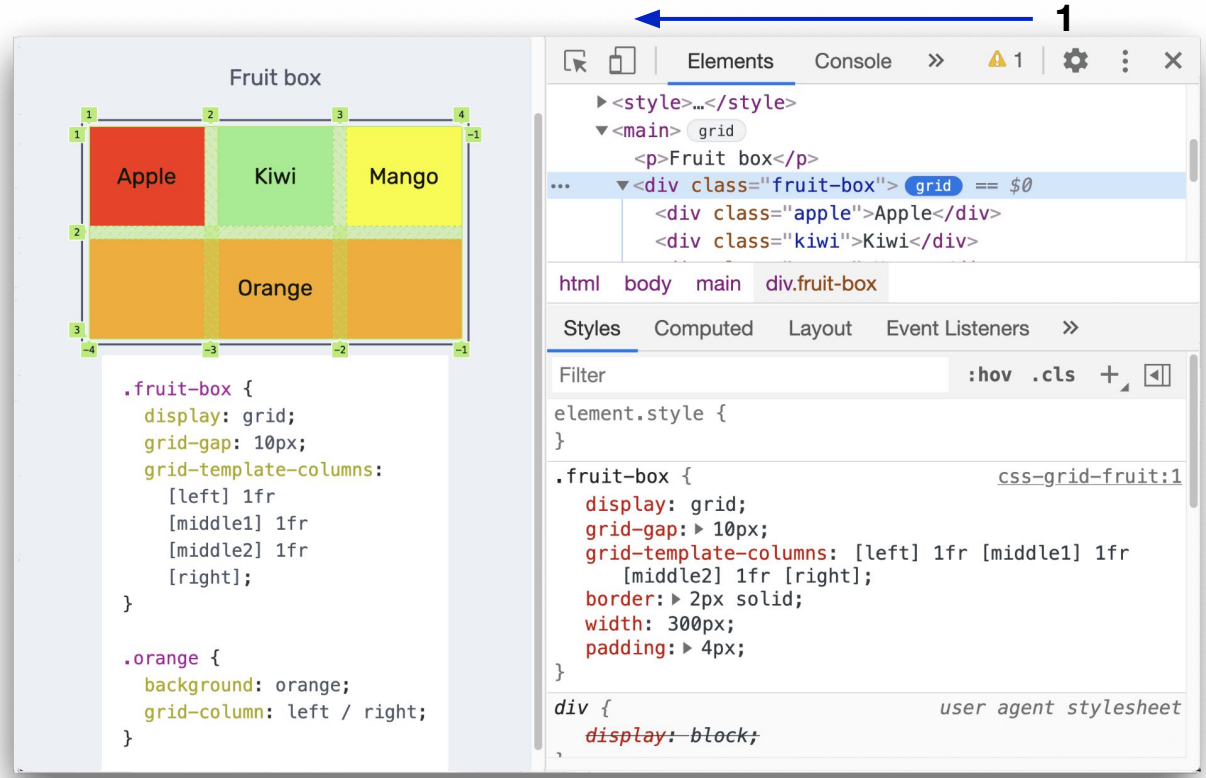
This grid has two equal-sized columns and uses automatically generated rows of 100 pixels height. The grid-auto-flow property is set to dense to fill in gaps and rows, so it first fills the rows.

Chrome Dev Tools

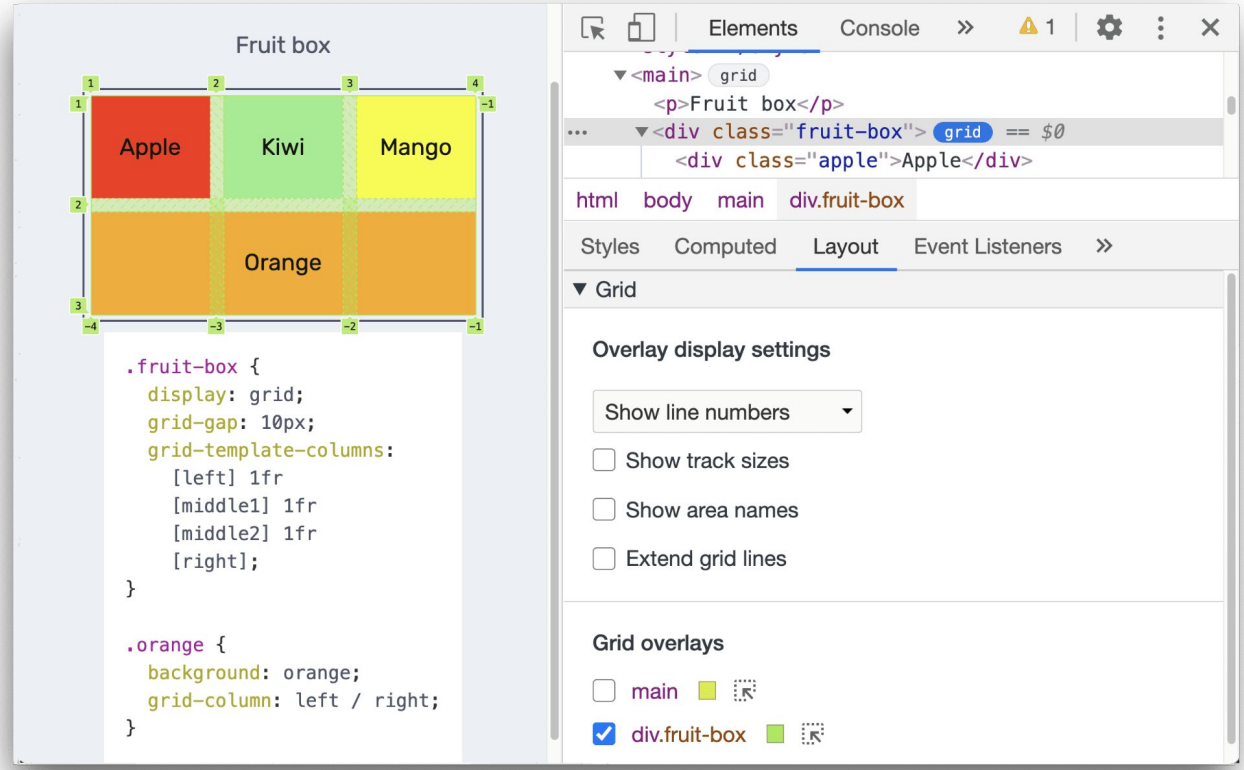
When an HTML element on your page has display: grid or display: inline-grid applied to it, you can see a grid badge next to it in the **Elements** panel.



Click the badge to toggle the display of a grid overlay on the page. The overlay appears over the element, laid out like a grid to show the position of its grid lines and tracks:



Open the **Layout** pane. When grids are included on a page, the Layout pane includes a **Grid** section containing a number of options for viewing those grids.

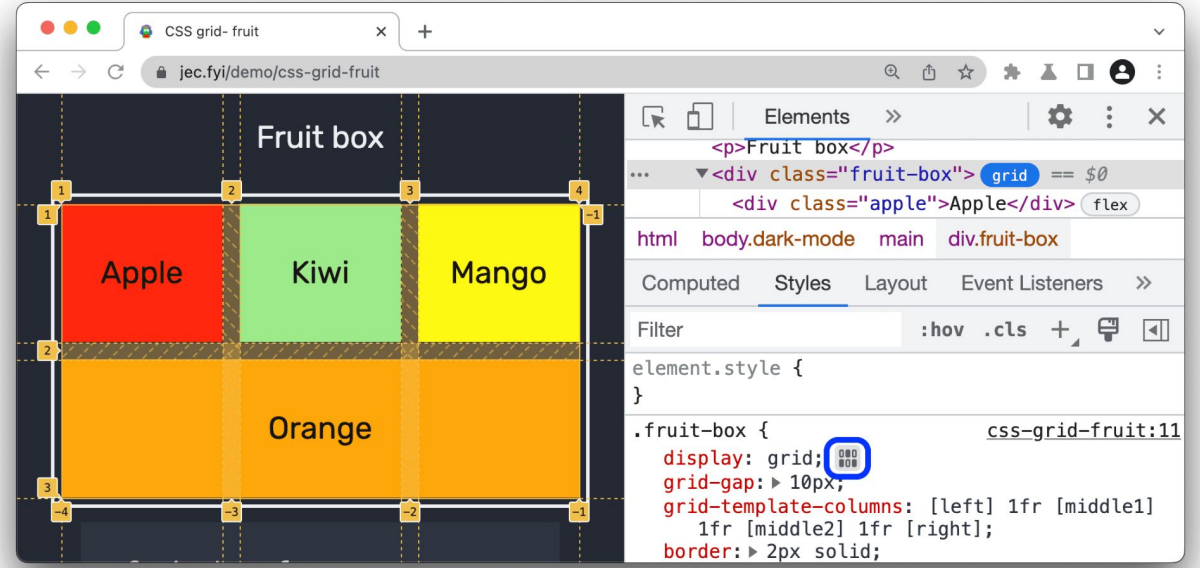


Align grid items and their content with the Grid Editor

You can align grid items and their content with a click of a button instead of typing CSS rules.

To align grid items and their content:

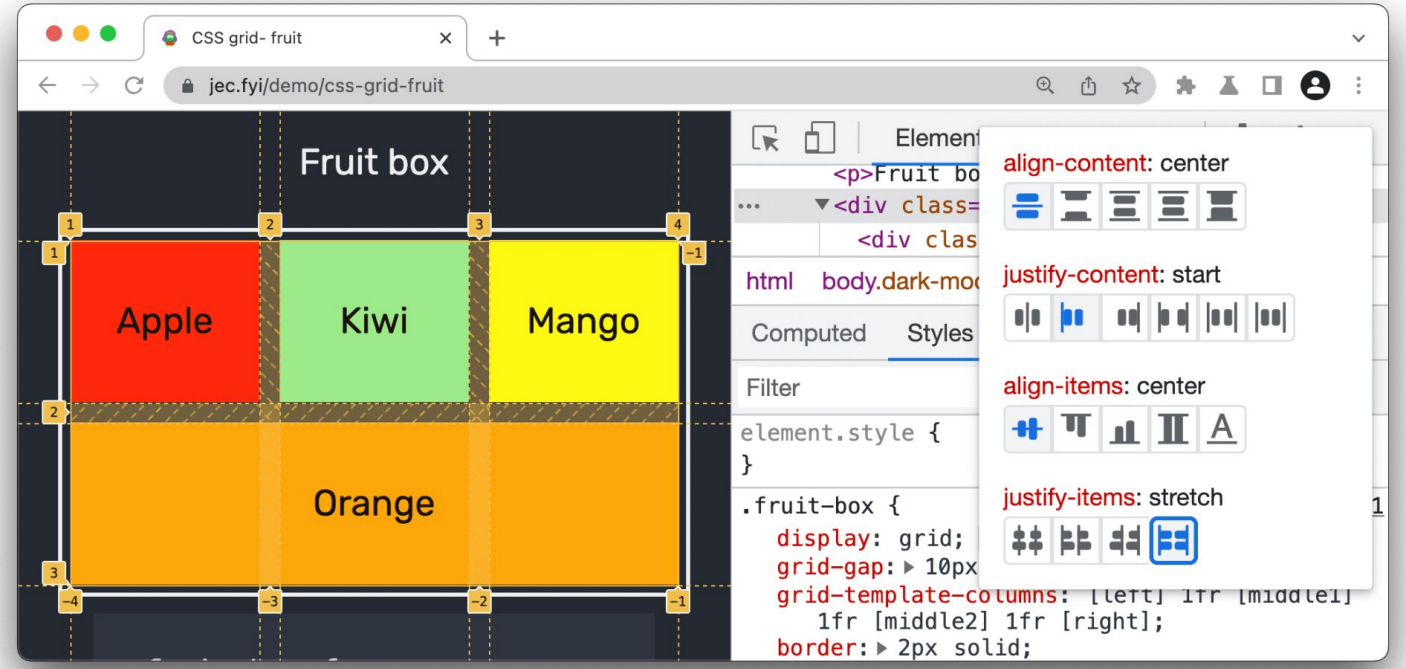
1. In the **Elements** > **Styles** pane, click the **Grid Editor** button next to display: grid.



In the **Grid Editor**, click the corresponding buttons to set the `align-*` and `justify-*` CSS properties for the grid items and their content.

2. Observe the adjusted grid items and content in the viewport.

3. Observe the adjusted grid items and content in the viewport.



Grid viewing options

The **Grid** section in the **Layout** pane contains 2 sub sections:

- Overlay display settings
- Grid overlays

Let's look into each of these sub sections in detail.

Overlay display settings

consists of two parts:

a. A drop-down menu with the following options:

- **Hide line labels**: Hide the line labels for each grid overlay.
- **Show line numbers**: Show the line numbers for each grid overlay (selected by default).
- **Show line names**: Show the line names for each grid overlay in the case of grids with line names.

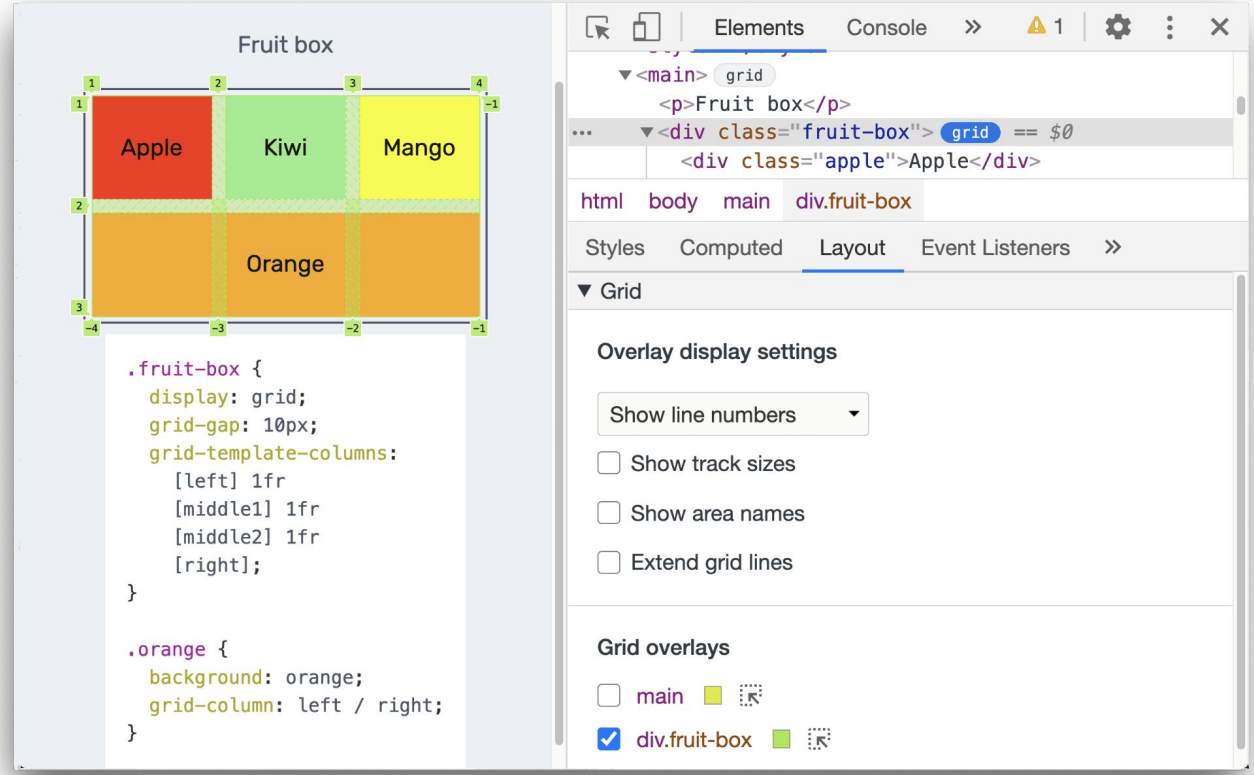
b. Checkboxes with options within:

- **Show track sizes**: Toggle to show or hide track sizes.
- **Show area names**: Toggle to show or hide area names, in the case of grids with named grid areas.
- **Extend grid lines**: By default, grid lines are only shown inside the element with `display: grid` or `display: inline-grid` set on it; when toggling this option on, the grid lines extend to the edge of the viewport along each axis.

Let's examine these settings in more detail.

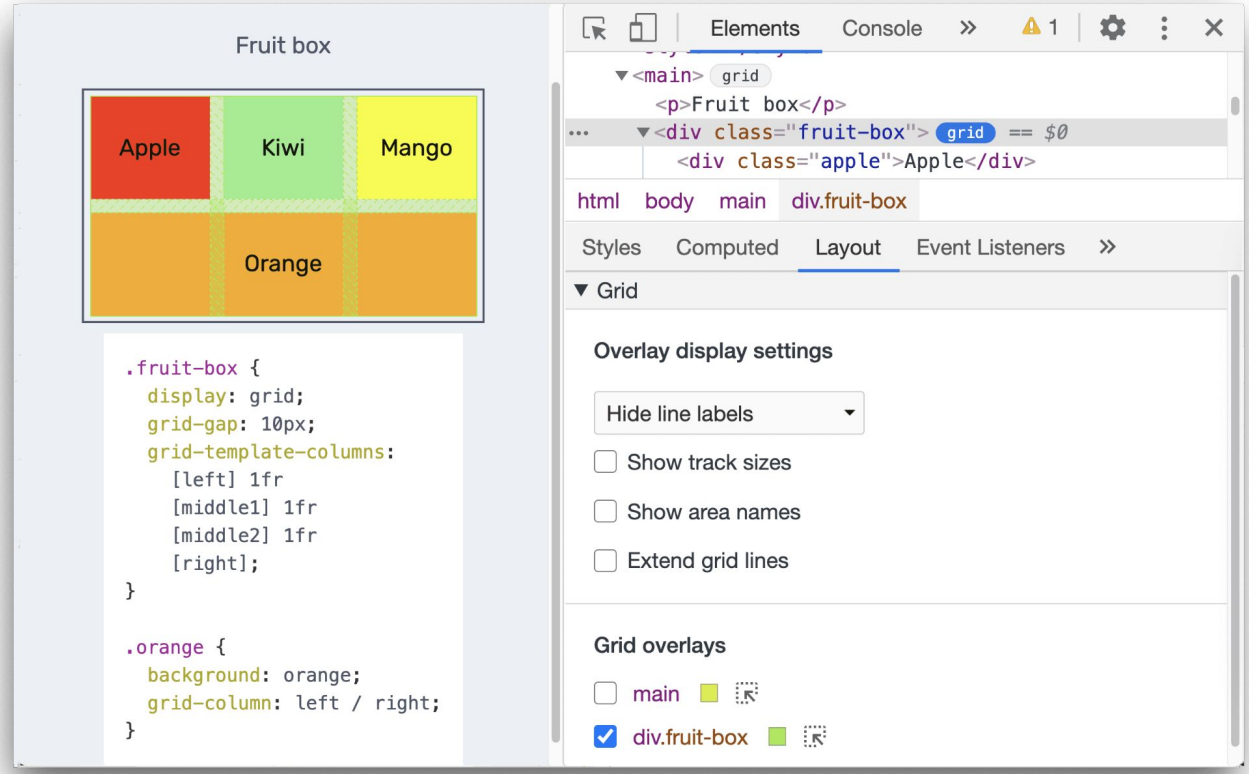
Show line numbers

By default, the positive and negative line numbers are displayed on the grid overlay.



Hide line labels

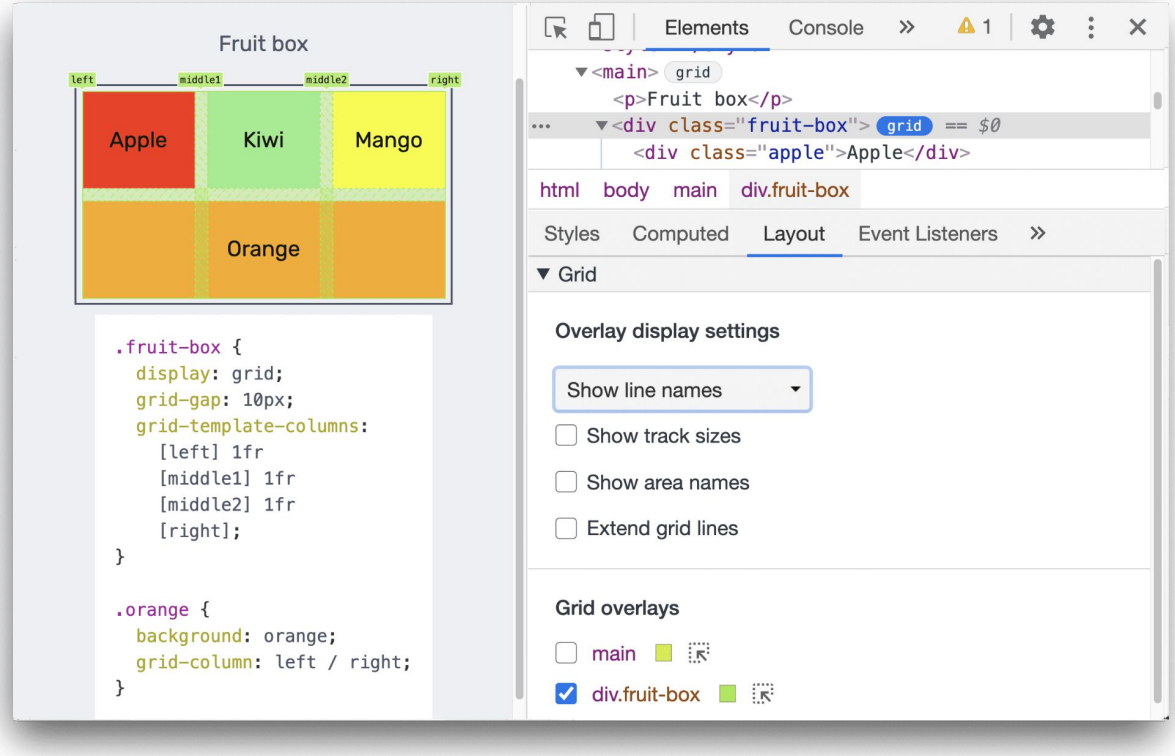
Select **Hide line labels** to hide the line numbers.



Show line names

You can select **Show line names** to view the line names instead of numbers. In this example, we have four lines with names: left, middle1, middle2 and right.

In this demo, the **orange** element spans from left to right, with CSS grid-column: left / right. Showing line names makes it easier to visualize the start and end position of the element.



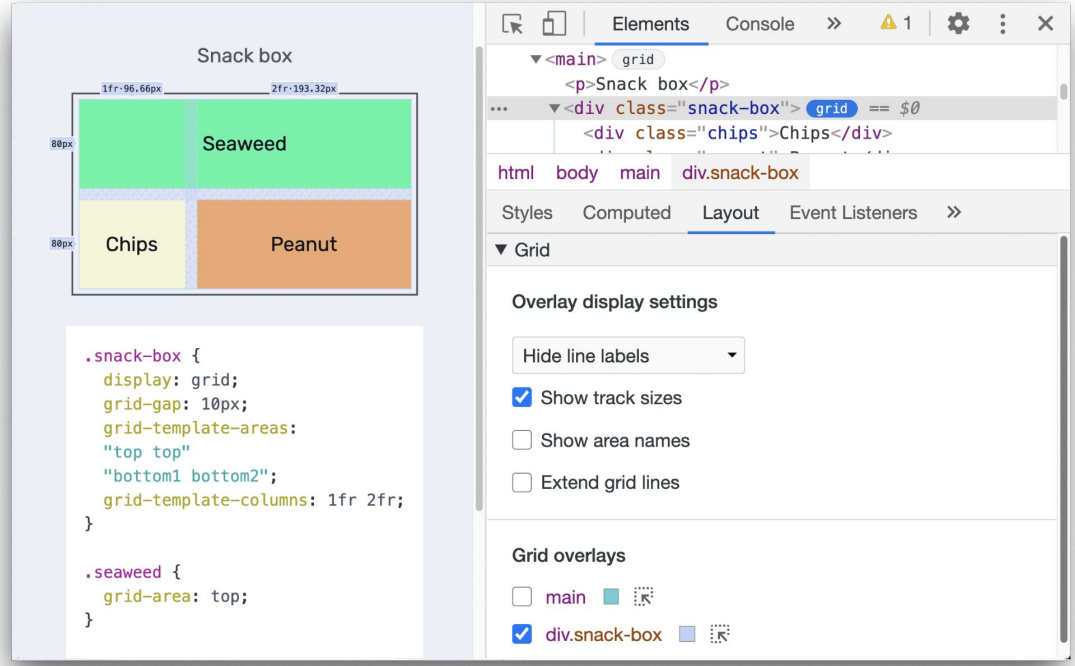
Show track sizes

Enable the **Show track sizes** checkbox to view the track sizes of the grid.

DevTools will display [authored size] - [computed size] in each line label: **Authored** size: The size defined in the stylesheet (omitted if not defined). **Computed** size: The actual size on the screen.

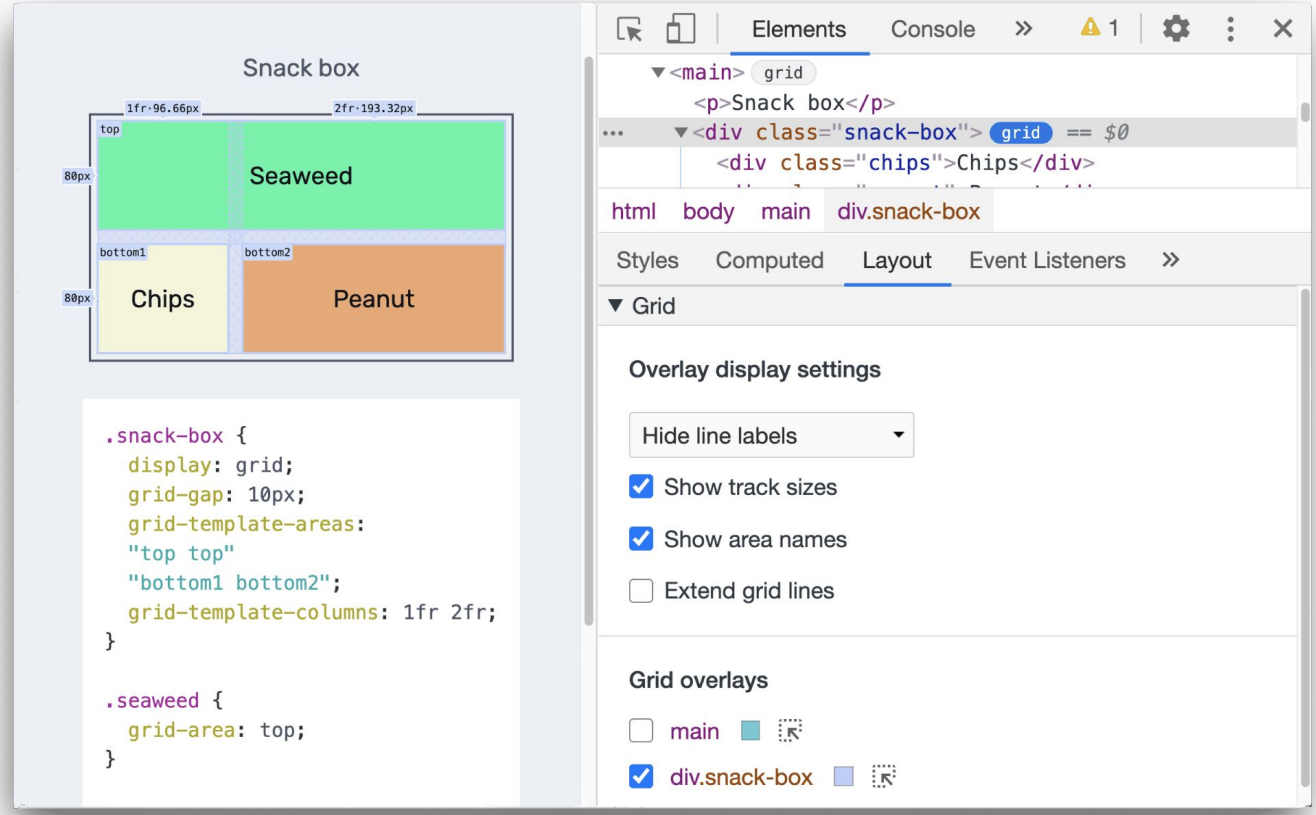
In this demo, the snack-box column sizes are defined in the CSS grid-template-columns:1fr 2fr;. Therefore, the column line labels show both authored and computed sizes: **1fr - 96.66px** and **2fr - 193.32px**.

The row line labels show only computed sizes: **80px** and **80px** since there are no row sizes defined in the stylesheet.



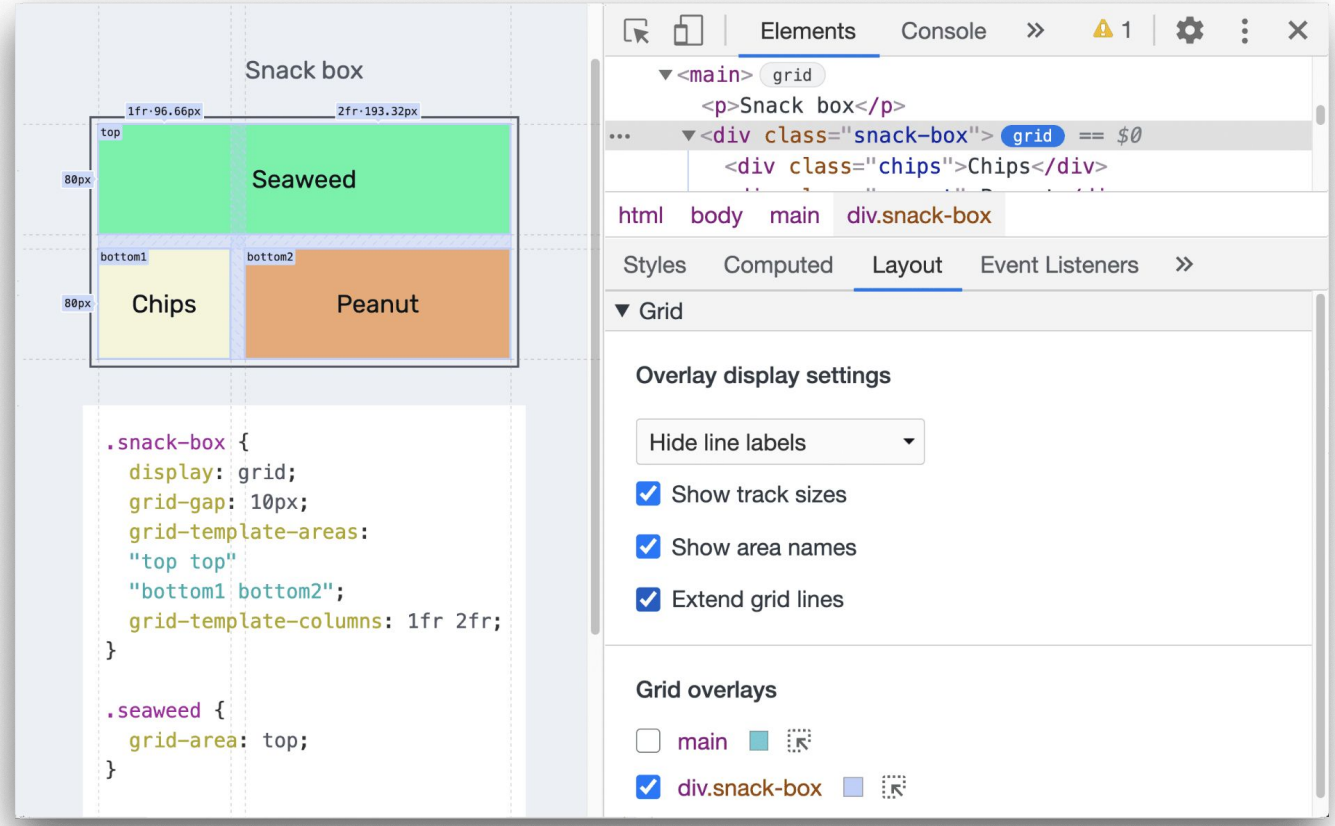
Show area names

To view the area names, enable the **Show area names** checkbox. In this example, there are three areas in the grid - **top**, **bottom1** and **bottom2**.



Extend grid lines

Enable the **Extend grid lines** checkbox to extend the grid lines to the edge of the viewport along each axis.

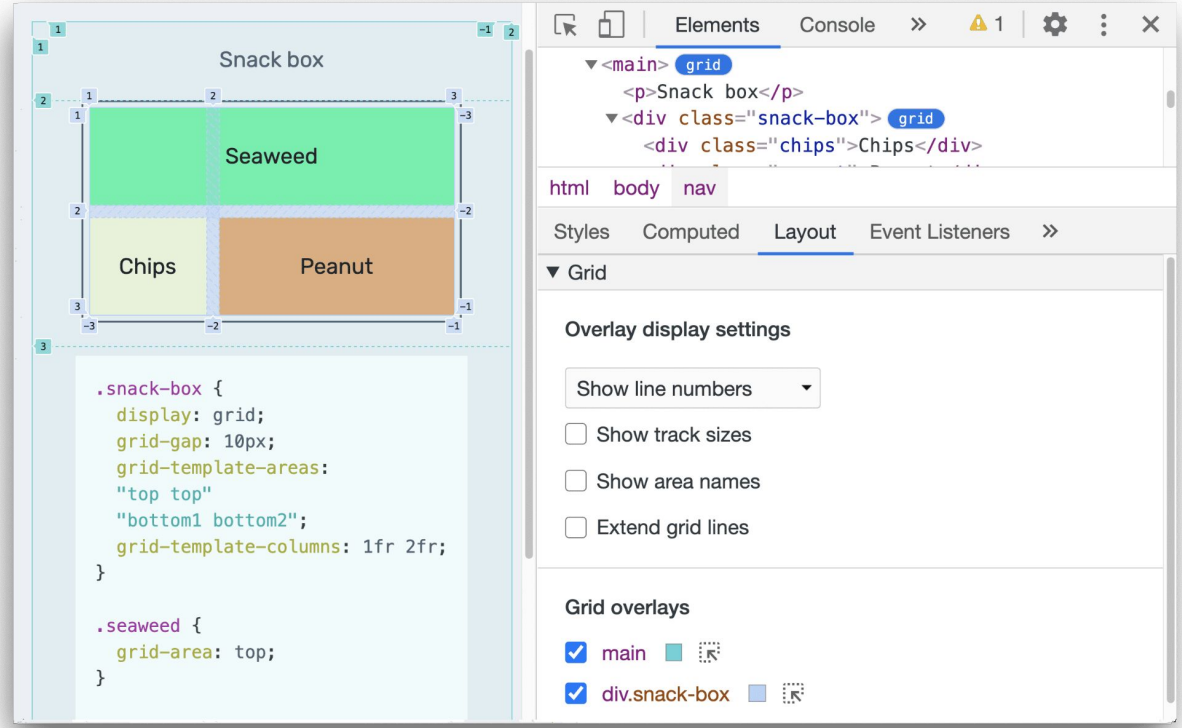


Grid overlays

The **Grid overlays** section contains a list of grids that are present on the page, each with a checkbox, along with various options.

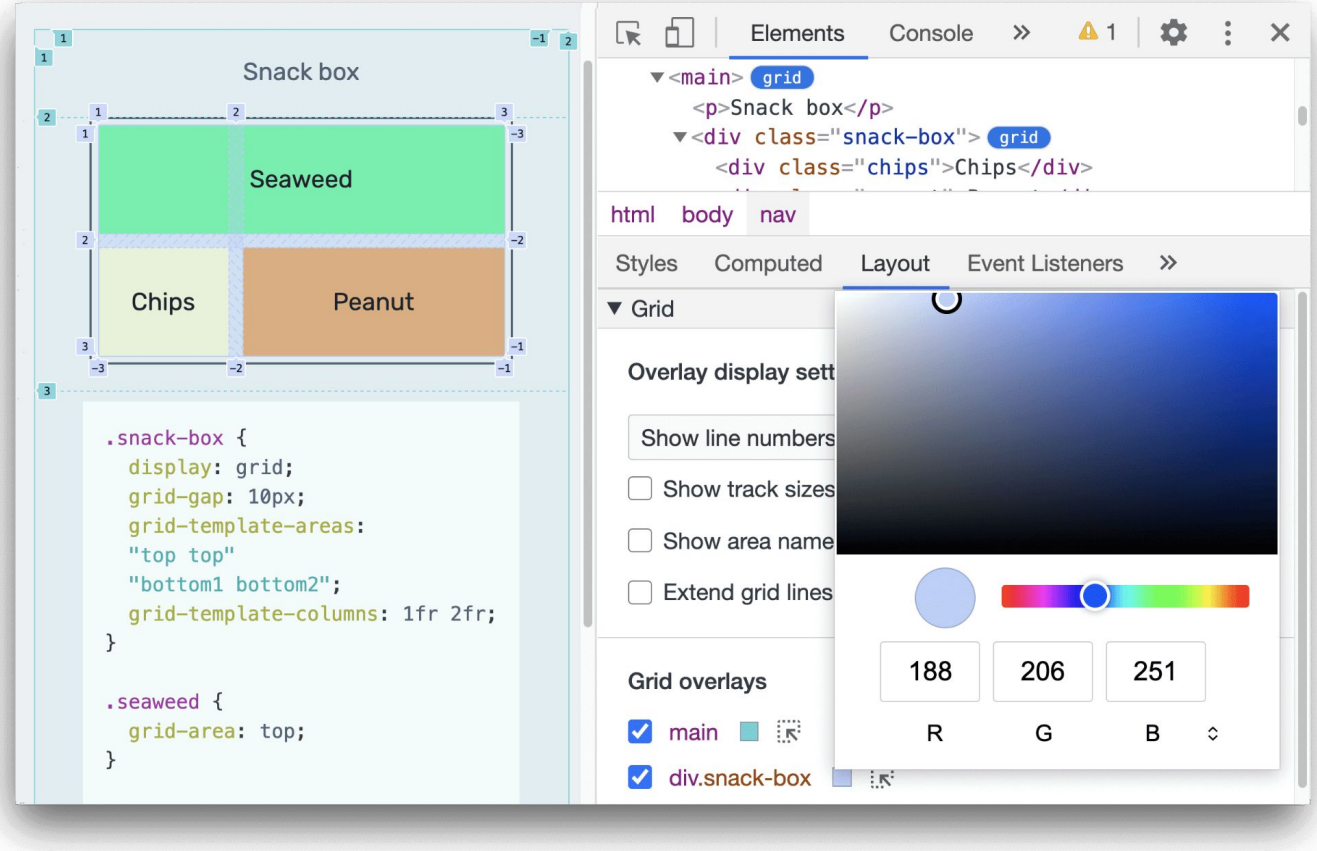
Enable overlay views of multiple grids

You can enable overlay views of multiple grids. In this example, there are two grid overlays enabled - main and div.snack-box, each represented with different colors.



Customize the grid overlay color

You can customize each grid overlay color by clicking the color picker.



Highlight the grid

Click the highlight icon to immediately highlight the HTML element, scroll to it in the page and select it in the Elements panel.