Relationship of grid layout to other layout methods

CSS Grid Layout has been designed to work alongside other parts of CSS, as part of a complete system for doing the layout. In this guide, I will explain how a grid fits together with other techniques you may already be using.

Grid and flexbox

The basic difference between CSS Grid Layout and CSS Flexbox Layout is that flexbox was designed for layout in one dimension - either a row *or* a column. Grid was designed for two-dimensional layout - rows, and columns at the same time. The two specifications share some common features, however, and if you have already learned how to use flexbox, the similarities should help you get to grips with Grid.

One-dimensional vs. two-dimensional layout

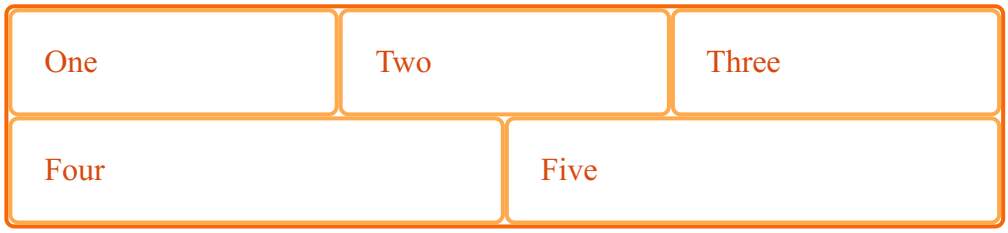
A simple example can demonstrate the difference between one- and two-dimensional layouts.

In this first example, I am using flexbox to lay out a set of boxes. I have five child items in my container, and I have given the flex properties values so that they can grow and shrink from a flex-basis of 150 pixels.

I have also set the `flex-wrap` property to `wrap`, so that if the space in the container becomes too narrow to maintain the flex basis, items will wrap onto a new row.

```
<div class="wrapper">
  <div>One</div>
  <div>Two</div>
  <div>Three</div>
  <div>Four</div>
  <div>Five</div>
</div>
```

```
.wrapper {
  width: 500px;
  display: flex;
  flex-wrap: wrap;
}
.wrapper > div {
  flex: 1 1 150px;
}
```



In the image, you can see that two items have wrapped onto a new line. These items are sharing the available space and not lining up underneath the items above. This is because when you wrap flex items, each new row (or column when working by column) becomes a new flex container. Space distribution happens across the row.

A common question then is how to make those items line up. This is where you want a two-dimensional layout method: You want to control the alignment by row and column, and this is where grid comes in.

The same layout with CSS grids

In this next example, I create the same layout using Grid. This time we have three `1fr` column tracks. We do not need to set anything on the items themselves; they will lay themselves out one into each cell of the created grid. As you can see they stay in a strict grid, lining up in rows and columns. With five items, we get a gap on the end of row two.

```
<div class="wrapper">
  <div>One</div>
  <div>Two</div>
  <div>Three</div>
  <div>Four</div>
  <div>Five</div>
</div>
```

```
.wrapper {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
}
```

A simple question to ask yourself when deciding between grid or flexbox is:

- do I only need to control the layout by row *or* column – use a flexbox
- do I need to control the layout by row *and* column – use a grid

Content out or layout in?

In addition to the one-dimensional versus two-dimensional distinction, there is another way to decide if you should use flexbox or grid for a layout. Flexbox works from the content out. An ideal use case for flexbox is when you have a set of items and want to space them out evenly in a container. You let the size of the content decide how much individual space each item

takes up. If the items wrap onto a new line, they will work out their spacing based on their size and the available space *on that line*.

Grid works from the layout in. When you use CSS Grid Layout you create a layout and then you place items into it, or you allow the auto-placement rules to place the items into the grid cells according to that strict grid. It is possible to create tracks that respond to the size of the content, however, they will also change the entire track.

If you are using flexbox and find yourself disabling some of the flexibility, you probably need to use CSS Grid Layout. An example would be if you are setting a percentage width on a flex item to make it line up with other items in a row above. In that case, a grid is likely to be a better choice.

Box alignment

The feature of flexbox that was most exciting to many of us was that it gave us proper alignment control for the first time. It made it easy to center a box on the page. Flex items can stretch to the height of the flex container, meaning that equal height columns were possible. These were things we have wanted to do for a very long time, and have come up with all kinds of hacks to accomplish, at least visually.

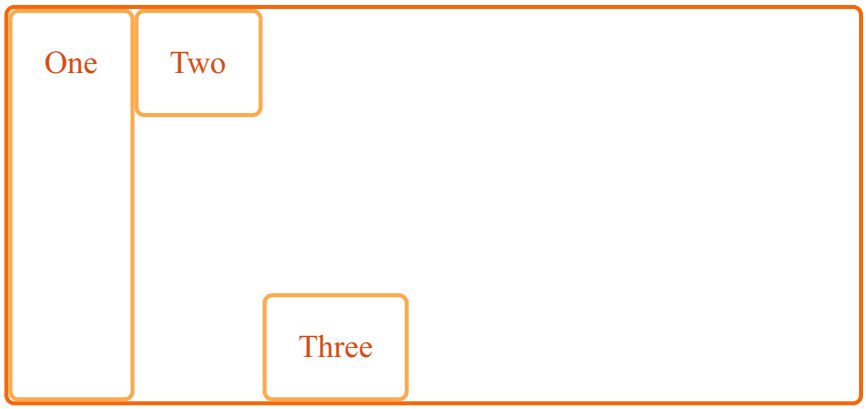
The alignment properties from the flexbox specification have been added to a new specification called Box Alignment Level 3. This means that they can be used in other specifications, including Grid Layout. In the future, they may well apply to other layout methods as well.

In a later guide in this series, I'll be taking a proper look at Box Alignment and how it works in Grid Layout. For now, here is a comparison between simple examples of flexbox and grid.

In the first example, which uses flexbox, I have a container with three items inside. The wrapper `min-height` is set, so it defines the height of the flex container. I have set `align-items` on the flex container to `flex-end` so the items will line up at the end of the flex container. I have also set the `align-self` property on `box1` so it will override the default and stretch to the height of the container and on `box2` so it aligns to the start of the flex container.

```
<div class="wrapper">
  <div class="box1">One</div>
  <div class="box2">Two</div>
  <div class="box3">Three</div>
</div>
```

```css
.wrapper {
  display: flex;
  align-items: flex-end;
  min-height: 200px;
}
.box1 {
  align-self: stretch;
}
.box2 {
  align-self: flex-start;
}
```



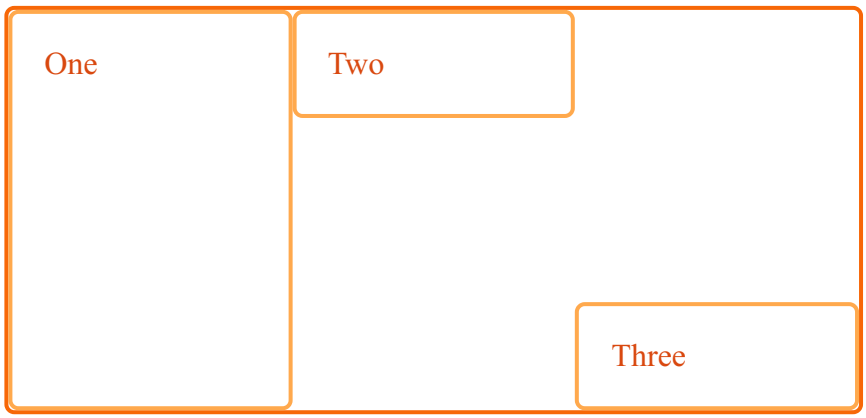
Alignment in CSS Grids

This second example uses a grid to create the same layout. This time we are using the box alignment properties as they apply to a grid layout. So we align to `start` and `end` rather than `flex-start` and `flex-end`. In the case of a grid layout, we are aligning the items inside their grid area. In this case that is a single grid cell, but it could be an area made up of several grid cells.

```html
<div class="wrapper">
  <div class="box1">One</div>
  <div class="box2">Two</div>
  <div class="box3">Three</div>
</div>
```

```css
.wrapper {
  display: grid;
```

```
    grid-template-columns: repeat(3,1fr);
    align-items: end;
    grid-auto-rows: 200px;
  }
  .box1 {
    align-self: stretch;
  }
  .box2 {
    align-self: start;
  }
```



The `fr` unit and `flex-basis`

We have already seen how the `fr` unit works to assign a proportion of available space in the grid container to our grid tracks. The `fr` unit, when combined with the `minmax()` function can give us very similar behavior to the `flex` properties in flexbox while still enabling the creation of a layout in two dimensions.

If we look back at the example where I demonstrated the difference between one and two-dimensional layouts, you can see there is a difference between the way of the two layouts work responsively. With the flex layout, if we drag our window wider and smaller, the flexbox does a nice job of adjusting the number of items in each row according to the available space. If we have a lot of space all five items can fit on one row. If we have a very narrow container we may only have space for one.

In comparison, the grid version always has three column tracks. The tracks themselves will grow and shrink, but there are always three since we asked for three when defining our grid.

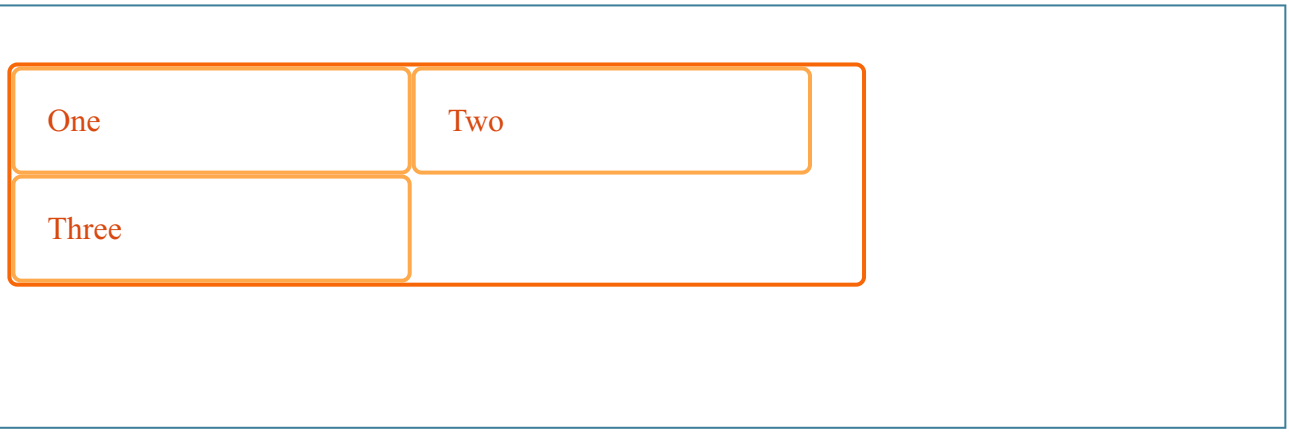
Auto-filling grid tracks

We can create a similar effect to flexbox, while still keeping the content arranged in strict rows and columns, by creating our track listing using repeat notation and the `auto-fill` and `auto-fit` properties.

In this next example, I have used the `auto-fill` keyword in place of an integer in the repeat notation and set the track listing to 200 pixels. This means that grid will create as many 200 pixels column tracks as will fit in the container.

```
<div class="wrapper">
  <div>One</div>
  <div>Two</div>
  <div>Three</div>
</div>
```

```
.wrapper {
  display: grid;
  grid-template-columns: repeat(auto-fill, 200px);
}
```



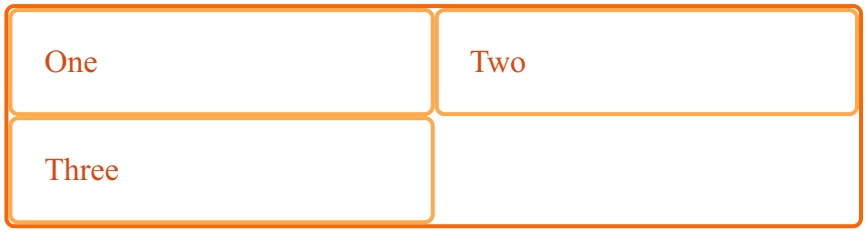
A flexible number of tracks

This isn't quite the same as flexbox. In the flexbox example, the items are larger than the 200 pixel basis before wrapping. We can achieve the same in grid by combining `auto-fill` and the `minmax()` function. In this next example, I create auto filled tracks with `minmax`. I want my tracks to be a minimum of 200 pixels, so I set the maximum to be `1fr`. Once the browser has worked out how many times 200 pixels will fit into the container–also taking account of grid

gaps–it will treat the `1fr` maximum as an instruction to share out the remaining space between the items.

```
<div class="wrapper">
  <div>One</div>
  <div>Two</div>
  <div>Three</div>
</div>
```

```
.wrapper {
  display: grid;
  grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));
}
```



We now have the ability to create a grid with a flexible number of flexible tracks, but see items laid out on the grid aligned by rows and columns at the same time.

Grid and absolutely positioned elements

Grid interacts with absolutely positioned elements, which can be useful if you want to position an item inside a grid or grid area. The specification defines the behavior when a grid container is a containing block and a parent of the absolutely positioned item.

A grid container as containing block

To make the grid container a containing block you need to add the position property to the container with a value of relative, just as you would make a containing block for any other absolutely positioned items. Once you have done this, if you give a grid item `position:`

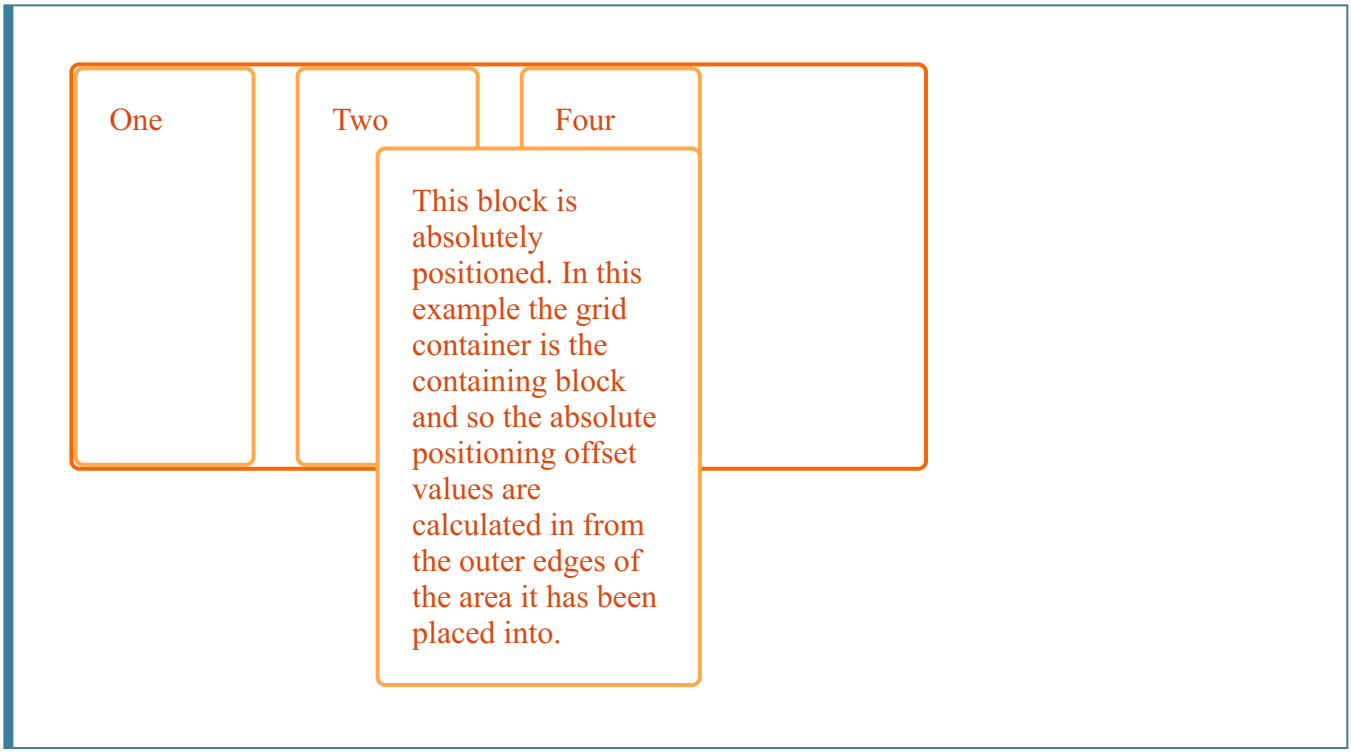
`absolute` it will take as its containing block the grid container or, if the item also has a grid position, the area of the grid it is placed into.

In the below example I have a wrapper containing four child items. Item three is absolutely positioned and also placed on the grid using line-based placement. The grid container has `position: relative` and so becomes the positioning context of this item.

```
<div class="wrapper">
  <div class="box1">One</div>

  <div class="box2">Two</div>
  <div class="box3">
   This block is absolutely positioned. In this example the grid container
  </div>
  <div class="box4">Four</div>
</div>
```

```
.wrapper {
  display: grid;
  grid-template-columns: repeat(4,1fr);
  grid-auto-rows: 200px;
  grid-gap: 20px;
  position: relative;
}
.box3 {
  grid-column-start: 2;
  grid-column-end: 4;
  grid-row-start: 1;
  grid-row-end: 3;
  position: absolute;
  top: 40px;
  left: 40px;
}
```

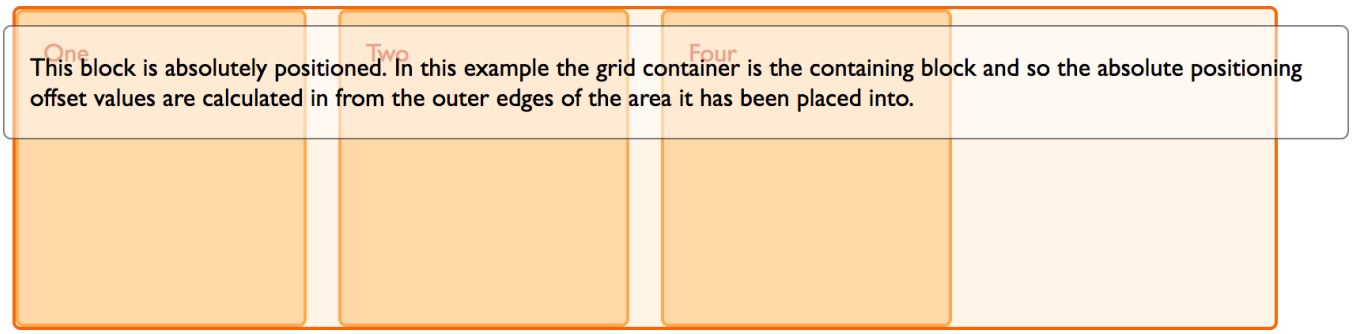


You can see that the item is taking the area from grid row line 2 to 4, and starting after line 1. Then it is offset in that area using the top and left properties. However, it has been taken out of flow as is usual for absolutely positioned items and so the auto-placement rules now place items into the same space. The item also doesn't cause the additional row to be created to span to row line 3.

If we remove `position: absolute` from the rules for `.box3` you can see how it would display without the positioning.

A grid container as parent

If the absolutely positioned child has a grid container as a parent but that container does not create a new positioning context, then it is taken out of flow as in the previous example. The positioning context will be whatever element creates a positioning context as is common to other layout methods. In our case, if we remove `position: relative` from the wrapper above, positioning context is from the viewport, as shown in this image.



Once again the item no longer participates in the grid layout in terms of sizing or when other items are auto-placed.

With a grid area as the parent

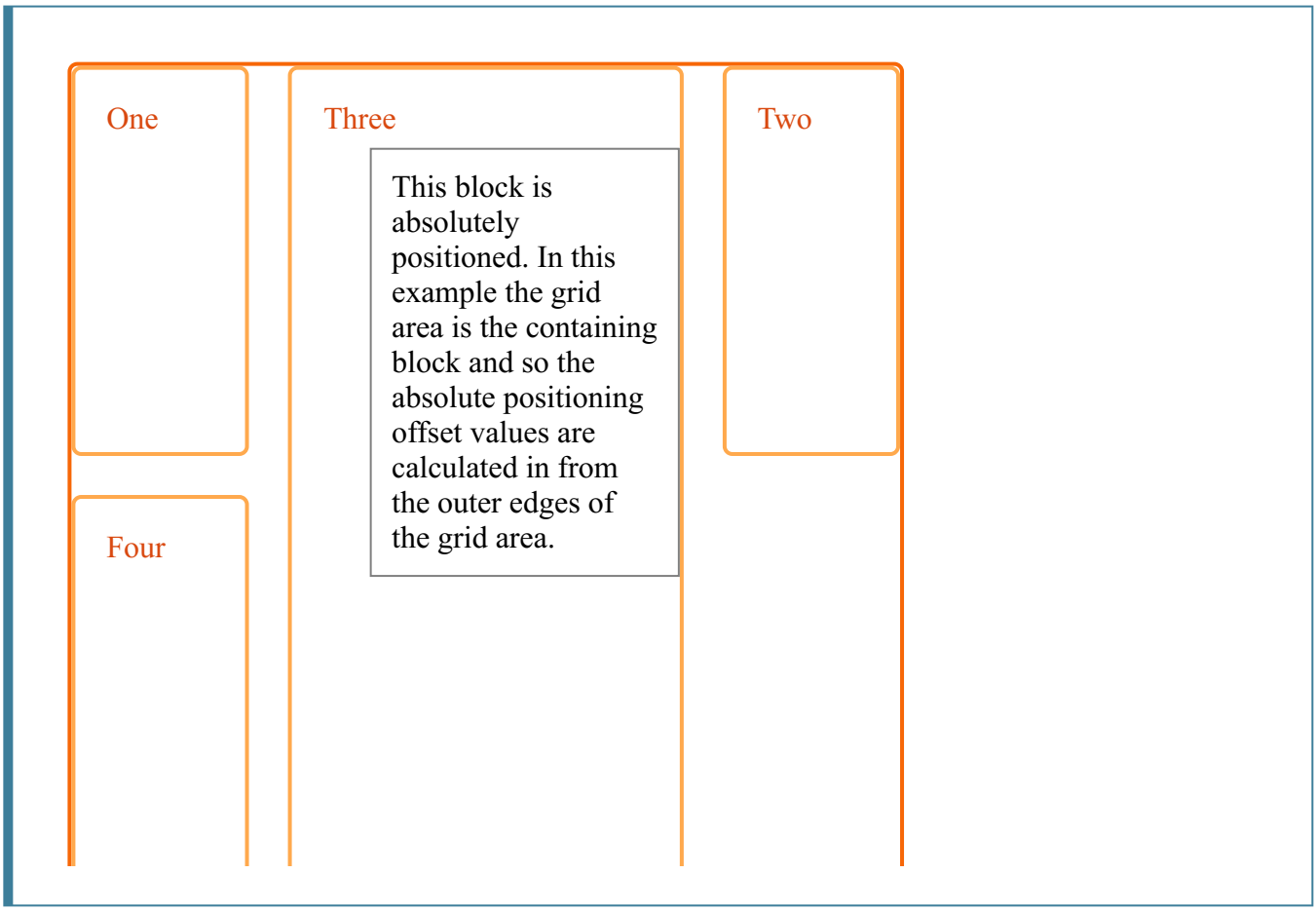
If the absolutely positioned item is nested inside a grid area then you can create a positioning context on that area. In the below example we have our grid as before but this time I have nested an item inside `.box3` of the grid.

I have given `.box3` position relative and then positioned the sub-item with the offset properties. In this case, the positioning context is the grid area.

```
<div class="wrapper">
  <div class="box1">One</div>
  <div class="box2">Two</div>
  <div class="box3">Three
    <div class="abspos">
     This block is absolutely positioned. In this example the grid area is
    </div>
  </div>
  <div class="box4">Four</div>
</div>
```

```
.wrapper {
  display: grid;
  grid-template-columns: repeat(4,1fr);
  grid-auto-rows: 200px;
  grid-gap: 20px;
}
.box3 {
  grid-column-start: 2;
  grid-column-end: 4;
  grid-row-start: 1;
  grid-row-end: 3;
  position: relative;
}
.abspos {
  position: absolute;
  top: 40px;
  left: 40px;
  background-color: rgba(255,255,255,.5);
```

```
    border: 1px solid rgba(0,0,0,0.5);
    color: #000;
    padding: 10px;
}
```



Grid and `display: contents`

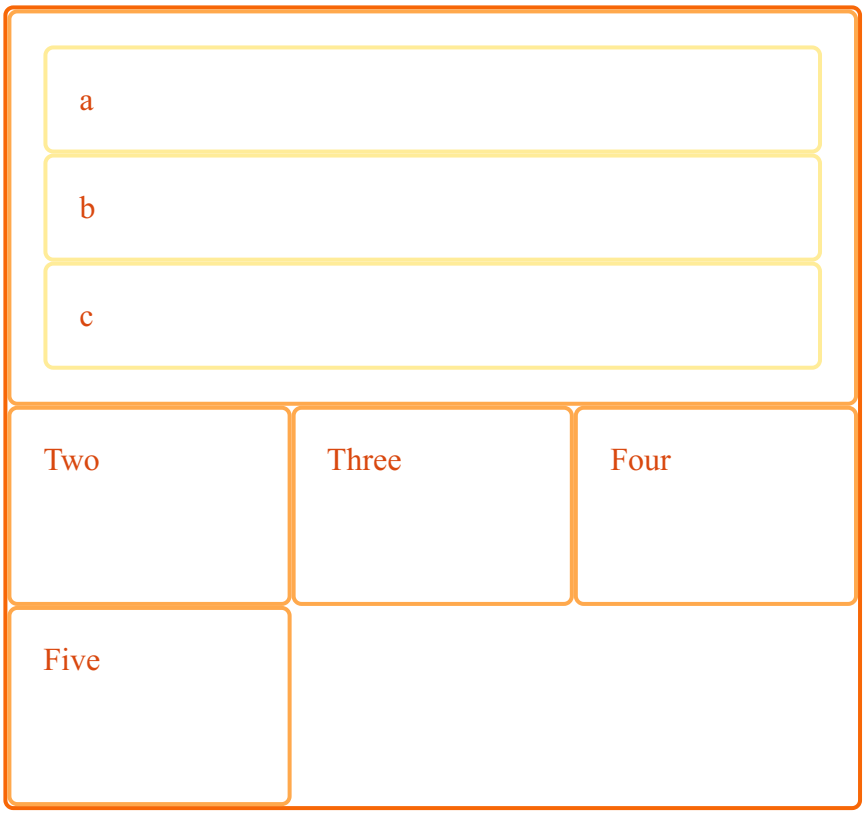
A final interaction with another layout specification that is worth noting is the interaction between CSS Grid Layout and `display: contents`. The `contents` value of the display property is a new value that is described in the Display specification as follows:

> “The element itself does not generate any boxes, but its children and pseudo-elements still generate boxes as normal. For the purposes of box generation and layout, the element must be treated as if it had been replaced with its children and pseudo-elements in the document tree.”

If you set an item to `display: contents` the box it would normally create disappears, and the boxes of the child elements appear as if they have risen up a level. This means that children of a grid item can become grid items. Sound odd? Here is a simple example. In the following markup, I have a grid and the first item on the grid is set to span all three column tracks. It contains three nested items. As these items are not direct children, they don't become part of the grid layout and so display using regular block layout.

```
<div class="wrapper">
  <div class="box box1">
    <div class="nested">a</div>
    <div class="nested">b</div>
    <div class="nested">c</div>
  </div>
  <div class="box box2">Two</div>
  <div class="box box3">Three</div>
  <div class="box box4">Four</div>
  <div class="box box5">Five</div>
</div>
```

```
.wrapper {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  grid-auto-rows: minmax(100px, auto);
}
.box1 {
  grid-column-start: 1;
  grid-column-end: 4;
}
```

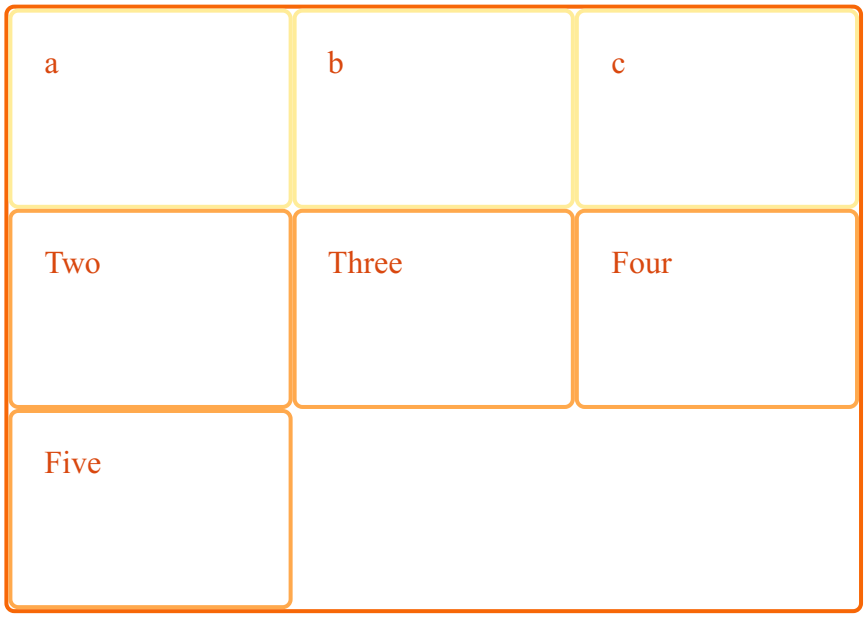


If I now add `display: contents` to the rules for `box1`, the box for that item vanishes and the sub-items now become grid items and lay themselves out using the auto-placement rules.

```
<div class="wrapper">
  <div class="box box1">
    <div class="nested">a</div>
    <div class="nested">b</div>
    <div class="nested">c</div>
  </div>
  <div class="box box2">Two</div>
  <div class="box box3">Three</div>
  <div class="box box4">Four</div>
  <div class="box box5">Five</div>
</div>
```

```
.wrapper {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  grid-auto-rows: minmax(100px, auto);
}
.box1 {
```

```
  grid-column-start: 1;
  grid-column-end: 4;
  display: contents;
}
```



This can be a way to get items nested into the grid to act as if they are part of the grid, and is a way around some of the issues that would be solved by subgrids once they are implemented. You can also use `display: contents` in a similar way with flexbox to enable nested items to become flex items.

As you can see from this guide, CSS Grid Layout is just one part of your toolkit. Don't be afraid to mix it with other methods of doing layout to get the different effects you need.

See Also

- Flexbox Guides
- Multiple-column Layout Guides