FLEXBOX

Sizing and dimensions

Default behavoir

The default sizing behavior depends upon the display property for an element.

Inline

Inline elements take the size of their content plus any padding. Additionally, inline elements *ignore* any explicit sizing properties (width, height, etc.) unless they are also position:absolute or position:fixed. This leads to a lot of confusion. If you have an inline element whose size you want to indicate explicitly, you should probably change it to inline-block.

Inline-block

Inline-block elements also take the size of their content, plus padding. However, they respect any explicit sizing properties. This is handy.

Block

By default when no sizing properties are used, block level elements take the width of their parent and the height of their content. Block level elements respect any explicit sizing properties.

Images

Images have an interesting behavior in that if only one dimension is set, the other is automatically calculated so that the original aspect ratio of the image is preserved. This is true for both decorative CSS images and <img> tags.

Sizing properties

There are six sizing properties. They are

- `width`
- `min-width`
- `max-width`
- `height`
- `min-height`
- `max-height`

The width and height properties are a simple way to explicitly set the width or height of an element. It is set directly, and the element maintains that dimension (unless it is inline and ignores these properties). And certainly, when it comes to images, there is little reason to pursue anything other than the simplest approach.

However, if you look back at the descriptions of the above inline-block and block level elements, you will notice that the inline-block elements are the size (height and width) of their content. So these elements are fundamentally **variably** sized, and this variability is one of the most powerful and useful aspects of these elements.

However, when we use an explicit width or height property, we remove that variability from the element. This makes it less powerful and less useful.

The min-width and min-height properties allow us to set a minimum boundary for that variability, but otherwise the variable sizing of the element is unimpeded. So if we have min-width:300px; that means the element will be 300 pixels or possibly wider.

Likewise the max-width and max-height properties allow us to set a maximum boundary for the variability.

As we move into flexbox based layouts, variability in our design will become very important.

Best Practice
Unless you have good cause, try to avoid using explicit dimension properties like width and height. If you must control the dimensions, consider using the min- or max- variants.

Overflow
The overflow property controls what happens to content that is too big to fit into an area.

https://www.w3schools.com/css/css_overflow.asp
https://codepen.io/paqui-molina/pen/OBYrXG

Box-sizing
The box-sizing property determines how the sizing properties are applied. It has two values: content-box and border-box.

The content-box value is default and simply means that the height or width properties affect the content box of the element and any padding or border is "additional".

When border-box is used, the sizing properties are used to set the "whole" size of the element, and the content size is likely to be less.

https://www.w3schools.com/css/css_boxmodel.asp

https://www.w3schools.com/css/css3_box-sizing.asp

Flexbox

Up to this point, we have covered quite a few different CSS layout concepts. Inline vs. block level display, different position values, various positioning properties, six different sizing properties, plus countless details and interactions. So by this time, we should know enough to make a two-column page design, right? Sadly, we cannot. The intrepid among us might be able to cobble something together by creatively using inline-block, or maybe absolute or fixed positioning, but ultimately, any design built with just the topics we've covered so far will likely be brittle or unwieldy. Why is that?

All the layout properties we've looked at have all applied to an *individual* element. But performing layout tasks like columnar layout or anything responsive requires coordinating *multiple* elements. This is where the flexbox comes in. When working with flexbox layout, there are some CSS properties that are applied to a parent element (also known as the flex container) and other CSS properties that are applied to the direct children of that parent (also known as the flex items). The flex container will handle laying out of its children. And, best of all, the flex container will lay out its children smartly, making the best use of the screen size available to it, while still following the general guidelines you laid down for it. As a general rule, layout with flexbox is pretty easy and the results are great.

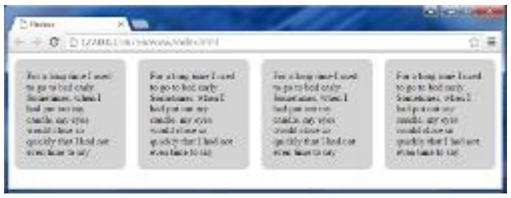
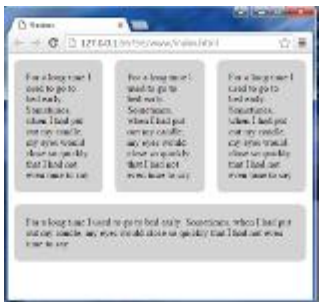
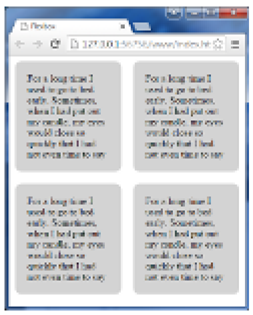
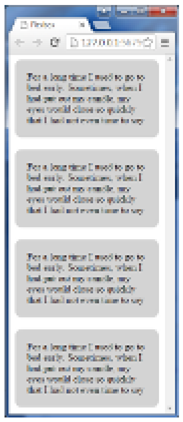
The **minimum** scenario for using flexbox is to make use of **two** CSS rules, and better results are achieved with a third.

display:flex;   /* on the container*/

flex:1   /* on the flex items*/

(better) flex-flow:row wrap; /*on the flex container*/

Here is a series of screen captures showing these minimum options applied to a parent <div> and four identical paragraphs at various browser sizes, with no other properties applied except some small margin and padding on the paragraph, and a background color and a border radius to help visualize.

For a long time I used to go to bed early. Sometimes, when I had put out my candle, my eyes would close so quickly that I had not even time to say https://codepen.io/paqui-molina/pen/GwEYJW

flex container

```
div  { display: flex; } /* block elements*/
span { display: inline-flex; }  /* inline elements*/
```

To designate an element as a flex container, we simply set the display property to be flex or inline-flex. A flex element will itself be a block level element, and an inline-flex element will itself be an inline element. However, in both cases the element is now a flex container and will be handling the layout of its children.

flex-flow

```
.fc {
   display: flex;
   flex-flow: row wrap;
 }
```

Flexbox containers can lay out their children both horizontally, as in a row, and vertically, as in a column, and *both at the same time*. This means that a single flex container not only can help you lay out a three column design, but also handle the header and footer above and below.

Strictly speaking, the flex-flow property is actually an abbreviation that replaces two other flexbox container properties: flex-direction and flex-wrap. But the row wrap value is so useful that it will likely be the standard.

```
flex-flow:<flex-direction> <flex-wrap>;
```

Values for the flex-direction: `row,  row-reverse`, `column`, `column-reverse`.

Values for the flex-wrap: `wrap`, `wrap-reverse`, `nowrap`

Flex items

The direct children of a flex container are automatically converted into flex items, with the exception of children that are position-fixed or position-absolute, which are taken out of the "flow" of the flex container. So there is no property needed to designate a child as a flex item, as it happens automatically.

One other automatic behavior to be aware of is that empty flex items are automatically removed from the flex container. Keep that in mind if you were planning on using an empty <div></div> construct as a placeholder for a CSS background image.

There is an array of flex item properties that can be applied to the children of a flex container, but there are three that we are not supposed to use in isolation. These three are: flex-grow, flex-shrink, and flex-basis. These three properties interrelate, so rather than using them in isolation the CSS3 specification encourages us to use the flex property, which can act as an abbreviation for all the three.

Flex property

Earlier, we saw that display:flex; can be used to designate a parent element as a flex container. In that case, the symbol "flex" is used as a value of the display property.

But flex is also the name of a property. It is a property that is applied to flex items, the children of a flex container.

```
span{flex:<flex-grow> <flex-shrink> <flex-basis>; }
```

The flex property provides a convenient way to abbreviate the three interrelated properties of flex-grow, flex-shrink, and flex-basis. The flex property *also* gives a flex item nice defaults for the optional properties. Therefore, flex:1; is **better** than flex-grow:1;.

Flex-grow

```
P {flex:1;/* rather than use flex-grow, use flex:<flex-grow>; */}
```

The flex-grow property is set simply to a positive number. In isolation that number means nothing. However, when the flex container is laying out its children, for any row (or column) it is processing it may end up with a little extra space. The flex-grow property determines how much extra space this flex item should get relative to its siblings. If one sibling has a flex-grow value of 2 and another -flex-grow value of 1, the former will receive twice as much of the extra space that is divided among the children.

A larger flex-grow value does not necessarily mean that an element is larger than its siblings that have smaller flex-grow values. The content of each sibling is first accounted for by the flex container when creating any row or column and only after that has been settled is any extra space distributed among the children.

Setting the flex-grow to 0 will prevent the flex item from growing. But remember, that will cause the item to shed its "flexible size" super-power.

Flex-shrink

```
p{flex:1 1; /* rather than use flex-shrink, use flex: <flex-grow> <flex-shrink>*/}
```

The flex-shrink is the opposite of flex-grow. When laying out any row or column, if the flex container needs to take away some space from the children, then those with the highest flex-shrink values contribute more of the needed space. Again, the flex-shrink value is just a number and it only has meaning when compared to its sibling flex-shrink values. And, again, this only occurs in the situation where the flex-container might need some space from its children.

Note: Like flex-grow, setting the flex-shrink to 0 will prevent the flex item from shrinking. However, this may *not* be as desirable as it first seems. Remember the box model from the previous section? If an item flex-shrink value is 0, then its border or padding may end up off-screen or pushed out of the parent, because there is a difference between "fitting" and "fitting nicely", and without the ability to be shrunk an item might fit but not "fit nicely". If you must set flex-shrink to 0, then it is recommended that you also set the box-sizing to border-box.

flex-basis

```
P {flex:1 1 87px; /* usa flex: <flex-grow> <flex-shrink> <flex-basis>*/}
```

The flex-basis can be used instead of the sizing properties on a flex item. If the flex-direction of the parent flex container is row or row-reverse, then the flex-basis will govern the width of the flex item. If the flex-direction is column or column-reverse, it governs the height.

The flex-basis provides the starting dimension (width or height) for the flex-item. It may be grown or shrunk from that. If you do not want it to change at all, then set the flex-grow and flex-shrink to 0, and the box-sizing to border-box. However, this is not advisable. Read the flex-shrink discussion above.

Best practices

Remember the minimum:

If you have a parent element that contains some child elements, then putting display:flex; on the parent is all that is needed to get started. The parent itself behaves like a normal block level element, it is the flex container, and all of its children are flex items.

It's not a bad idea to specify the flex-flow for the flex container (flex-flow: row wrap;), but it isn't required.

It's generally a good idea to also initialize the flex property on the flex items (e.g. flex:1), but again, this isn't required.

Use variable dimensions on flex items instead of explicit ones

This advice may not apply to images and may not be appropriate for every flex item. However, for most flex items, try to avoid using explicit width and height properties. Instead, use the flex-basis to set a desired dimension (e.g. flex: 1 1 **200px**;). Or consider using min-width (or max-width) and min-height (or max-height).

Doing so will make your flex items a bit more malleable. In CSS professional parlance, this is called being "responsive".

Do not over-constrain your flex items. Let the browser work for you.

Thinking of using inline-block? Consider flexbox instead.

Centering? Maybe flexbox
If you need to center some content horizontally, then the previous section on centering may help. It discusses the various options for inline or block level elements. However, a flex container and a flex child is also a possible approach. The flex container property justify-content:center; might help. Both approaches (flexbox and "traditional") have their advantages in various situations. Use your judgment.

If you need to center something vertically, unless it is an inline element or the content of a table cell, the best answer is almost certainly flexbox. Use the align-content and justify-content properties.

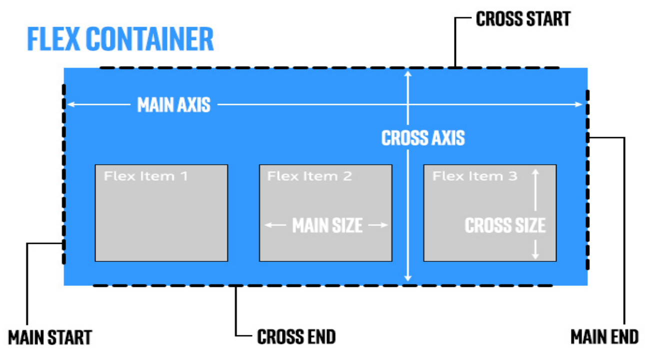
AVOID margin: auto on flex items

margin: auto prevents align-self from working.

Main and cross axes

Every flexbox has two axes. The "main" axis is the major axis along which all the flex items are being laid. So, when the flex-direction is row, the main axis is horizontal, running left to right. When the flex-direction is column, the main axis is vertical, running top to bottom.

The "cross" axis runs perpendicular to the main axis. It is the direction that items might wrap. So with flex-flow: row wrap; the cross axis is vertical and runs top to bottom. And with flex-flow:row wrap-reverse; it is vertical running bottom to top.



In the illustration above, we see the main and cross axes as they would be for flex-flow:row wrap;. And in the same illustration, we also see the start and end points for both the main and cross axes. Take a moment and visualize how both the axes *and* the start and end points would change for each of these combinations for flex-flow:

https://codepen.io/paqui-molina/pen/WYZwNO

- `row wrap`
- `row wrap-reverse`
- `row-reverse wrap`
- `row-reverse wrap-reverse`
- `column wrap`
- `column wrap-reverse`
- `column-reverse wrap`
- `column-reverse wrap-reverse`

main axis for sizing, cross axis for alignment

The terms *"main"* and *"cross"* appear in the descriptions of flexbox and in multiple online tutorials you might find. However, they are **not** used in the names of any CSS properties or values. The following properties control behavior along the main axis. We are already familiar with all of them, except justify-content.

- `flex`
- `flex-grow`
- `flex-shrink`
- `flex-basis`
- `justify-content`

All the properties above control how a flex item might be *sized*, except justify-content, which controls how a flexbox container spaces out and positions flex items.

And, the following list of properties controls behavior along the cross axis:

- `align-content`
- `align-items`
- `align-self`

These properties all govern how a flex item might be *aligned* or positioned along the cross axis. They also support a simple stretch value, which we will see when we cover them.

Important: Flexbox items can have their size and position influenced on the main axis, with flex-grow, and others. But on the cross axis, with the exception of a coarse stretch option, we can only influence their position. This distinction is very important. In the cross axis direction, our ability to influence the sizing is limited. When not using stretch, we get the normal sizing behavior (block level elements take vertical size of content, etc.) or use the regular sizing properties (min-height, max-height, height, etc.)

flex-start and flex-end are contextual

We will begin to see the values flex-start and flex-end in the next section as we look at alignment and justification. Just as the terms "main" and "cross" do not appear as CSS terms, be aware that flex-start and flex-end are contextual. When used with the justify-content property, flex-start and flex-end refer to the "main start" and "main end" sides (as in the illustration above). When used with any of the flexbox align properties, flex-start and flex-end refer to the "cross start" and "cross end" sides.

Justification and alignment

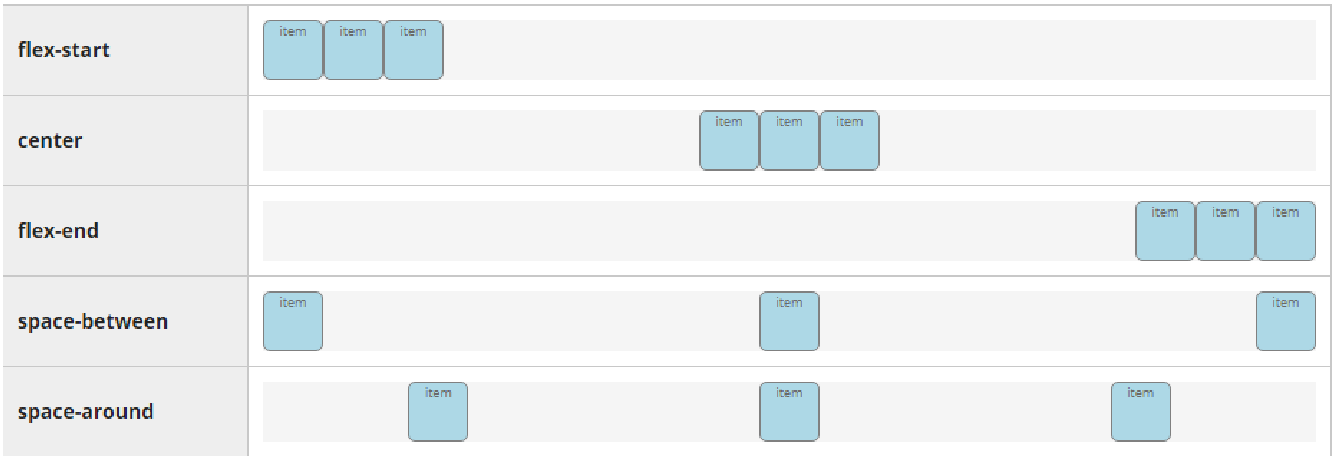
justify-content

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

When all the flex items in a flexbox container are fully resizable, then there will not be any extra space to put between the items. However, when the flex items are fixed size, or cannot grow anymore, then the flexbox container will put the extra space between or outside the items. The closest analogue from typography is called "justifying". Thus, the justify-content property serves a similar function for flexbox containers.

The justify-content property is applied to the flex container. It governs how any extra space along the main layout axis is distributed between the flexbox items. The possible values are: flex-start, flex-end, center, space-between, and space-around. The flex-start, flex-end, and center values do not distribute any space between the flex items. Instead, these values determine where the flex items should be positioned within the flex container, and any extra space is outside them. The space-between and space-around values both put space evenly between the flex items, but space-between places the flex items flush against the main start and main ends of the flexbox container. Remember that this is only spacing in the direction of the main axis. Justify-content does not affect any spacing or placement in the direction of the cross axis.

The table below should help illustrate this. It shows the justification options for a flexbox container with flex-flow:row;



If the flex-direction were row-reverse, then the only thing to change in the table above would be that the appearances of flex-start and flex-end would be reversed.

If the flex-direction were column, then remember that, if the flexbox container is a block level element, its default size would be that of its content. Which means there would be no extra vertical space to distribute. So all the five options above would be identical (a tight stack of items) *unless* the height of the flexbox container were explicitly made larger.

align-content and align-items

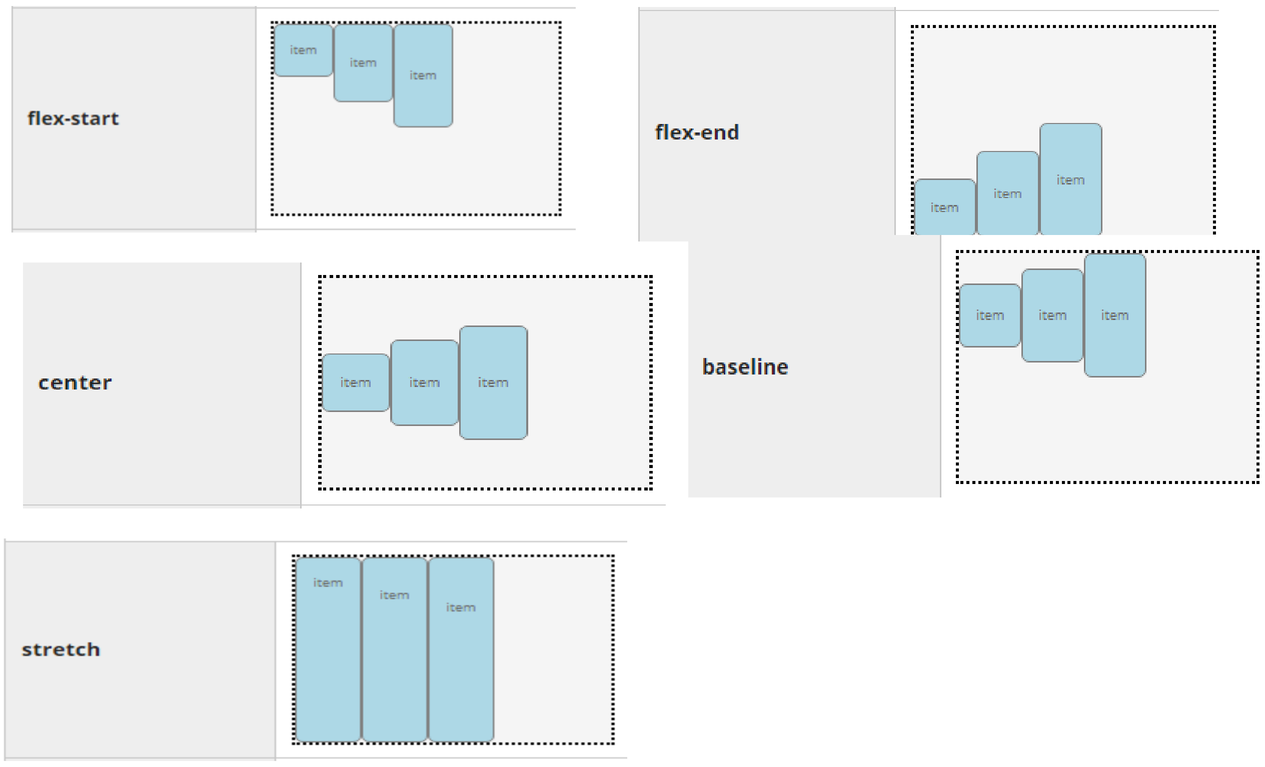
The align-content and align-items are often confused for one another. But they are very different. Both properties only apply if there is extra space in the cross axis direction. This is important to remember, because in many situations there isn't any cross axis space. In the example above (used for justify-content), none of the flexbox containers has any extra cross axis space (vertical space).

align-items

.fc {align-items:stretch; }

The align-items determines how items are aligned in the cross axis direction. This is applied to the flexbox container. The possible values are stretch, flex-start, flex-end, center, and baseline. In the context of alignment, flex-start and flex-end refer to the cross start and cross end sides, which may be swapped if the flex-wrap:wrap-reverse option is elected. Align-items defaults to stretch, if it is not set. The table below should help illustrate this. It shows a flex container with flex-flow:row; and a min-height value that is greater than the height of any of the items.

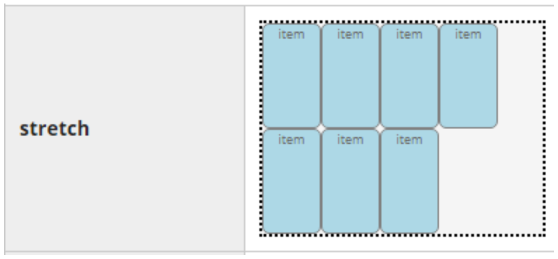
In the example below, each item has a different line-height value, so you can see how they align to each other.

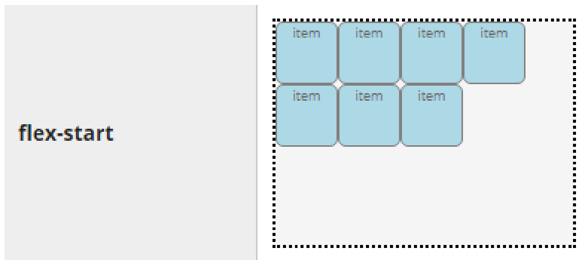


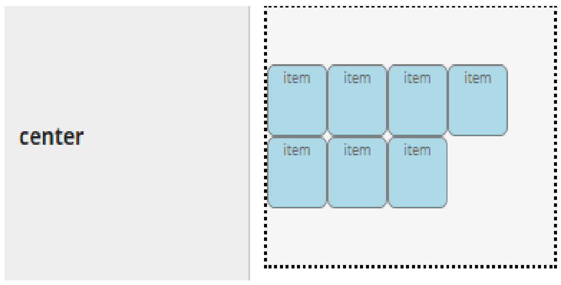
La propiedad strecth no tendrá ningún sentido si los flex items tienen un tamaño fijo. Si queremos que tenga efecto debemos usar : min-width o min-height .

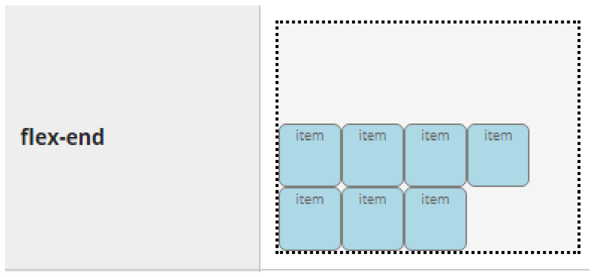
`align-content`
`.fc{align-content:space-between; }`

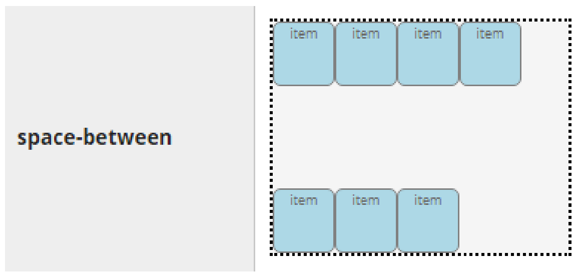
align-content is only relevant when the flexbox container supports wrapping and the flex items are, in fact, wrapping. The align-content determines how the *wrapped lines* are positioned or spaced. Align-content is not applied to individual items, but rather to the wrapped lines. The align-content property supports the stretch value, as well as the same values as justify-content (flex-start, center, flex-end, space-between, space-around). This is easier to understand from an example. Below we have a flex container with flex-flow:row wrap; and a height value that is greater than the height of any of the items.

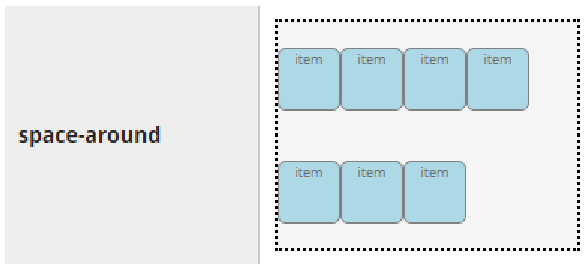






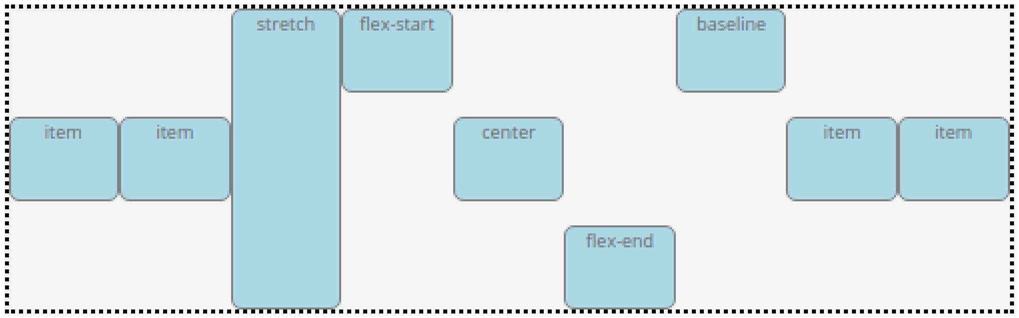






Igual que en la propiedad anterior La propiedad strecth no tendrá ningún sentido si los flex items tienen un tamaño fijo. Si queremos que tenga efecto debemos usar : min-width o min-height .

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



Unlike the other flexbox align properties, align-self is applied to an individual flex item, not to a flexbox container. This allows an individual flex item to be aligned differently than its siblings. Again, this is only true for cross axis alignment, and will only come into play, if there is extra space in the cross axis direction to be exploited.

The values for align-self are stretch, flex-start, center, flex-end, and baseline.

Align-self is ignored, if any of the four margins on the item is set to auto.

In the example below, we have a flex container with flex-flow:row; and align-items:center;. The individual items have their align-self property set.

Order

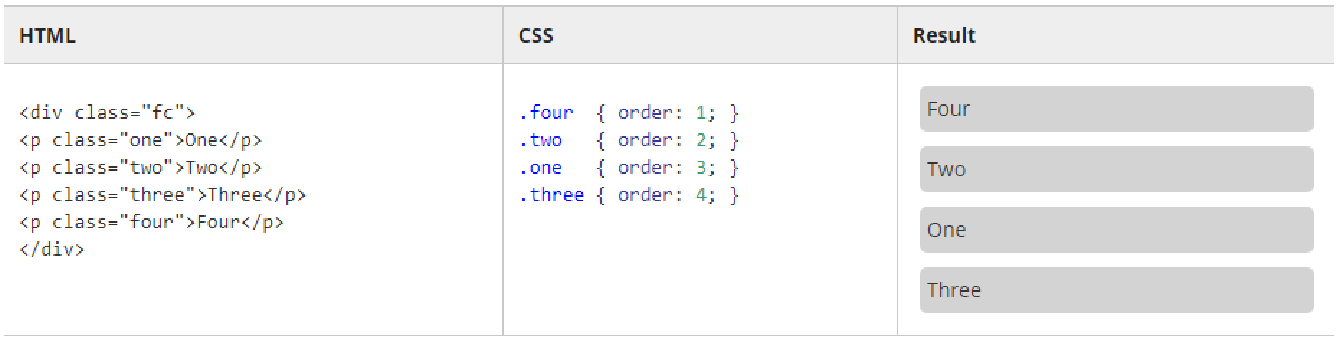
Interesting and simple property.

```
.item{order:2; }
```

The order property allows you to determine the order in which the item appears in the flexbox. This allows you to present the information in the flexbox layout independent of its order in the HTML itself. This is very useful, as there are many factors competing to drive the order of an HTML file.

For example, semantically, for SEO (Search Engine Optimization), and for accessibility for disabled visitors, the best practice is to put the title of an article before the article itself. While that seems simple enough, if your flexbox layout uses a flex-direction value of column-reverse, this could be a problem.

The order property, when applied to an individual flexbox item, lets you set its order. By default, the first item in a flexbox container has the order value of 1, the second is 2, etc. And you can override it.



HTML	CSS	Result

`<div class="fc">`
`<p class="one">One</p>`
`<p class="two">Two</p>`
`<p class="three">Three</p>`
`<p class="four">Four</p>`
`</div>`
	`.four  { order: 1; }`
	`.two   { order: 2; }`
	`.one   { order: 3; }`
	`.three { order: 4; }`
		Four
		Two
		One
		Three

Examples:

justify-content

Menú vertical con display:block

Menú vertical con flexbox: https://codepen.io/paqui-molina/pen/MWeNWBL

Align-content

Align-items and align-self

Align-items:baseline

Order