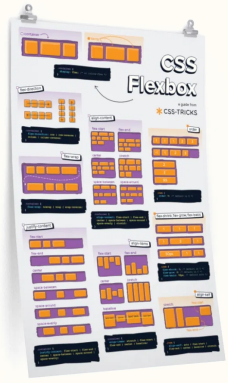
A Complete Guide to Flexbox

Chris Coyier Apr 8, 2013 Sep 10, 2021

Our comprehensive guide to CSS flexbox layout. This complete guide explains everything about flexbox, focusing on all the different possible properties for the parent element (the flex container) and the child elements (the flex items). It also includes history, demos, patterns, and a browser support chart.

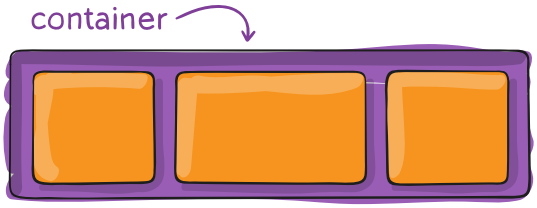


(/product/css-flexbox-poster/)

(#background) Background

(#basics-and-terminology) Basics and terminology

(#flexbox-properties) Flexbox properties



(#properties-for-the-parentflex-container) Properties for the Parent (flex container)

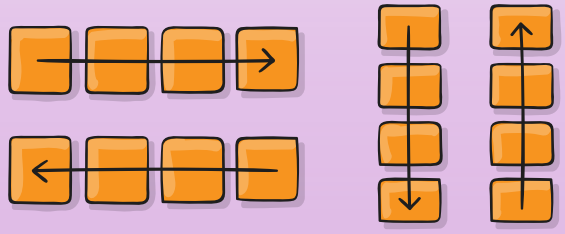
(#display) display

This defines a flex container; inline or block depending on the given value. It enables a flex context for all its direct children.

```css
.container {
  display: flex; /* or inline-flex */
}
```

Note that CSS columns have no effect on a flex container.

(#flex-direction) flex-direction

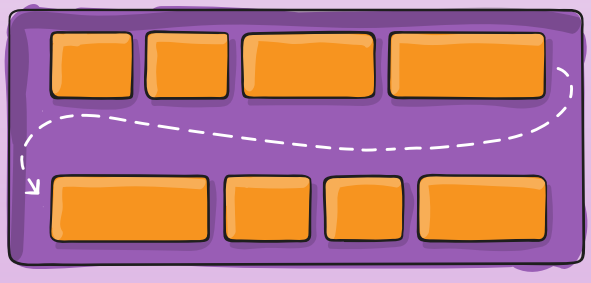


This establishes the main-axis, thus defining the direction flex items are placed in the flex container. Flexbox is (aside from optional wrapping) a single-direction layout concept. Think of flex items as primarily laying out either in horizontal rows or vertical columns.

```css
.container {
  flex-direction: row | row-reverse | column | column-reverse;
}
```

- `row` (default): left to right in `ltr`; right to left in `rtl`
- `row-reverse`: right to left in `ltr`; left to right in `rtl`
- `column`: same as `row` but top to bottom
- `column-reverse`: same as `row-reverse` but bottom to top

(#flex-wrap) flex-wrap



By default, flex items will all try to fit onto one line. You can change that and allow the items to wrap as needed with this property.

```css
.container {
  flex-wrap: nowrap | wrap | wrap-reverse;
}
```

- `nowrap` (default): all flex items will be on one line
- `wrap`: flex items will wrap onto multiple lines, from top to bottom.
- `wrap-reverse`: flex items will wrap onto multiple lines from bottom to top.

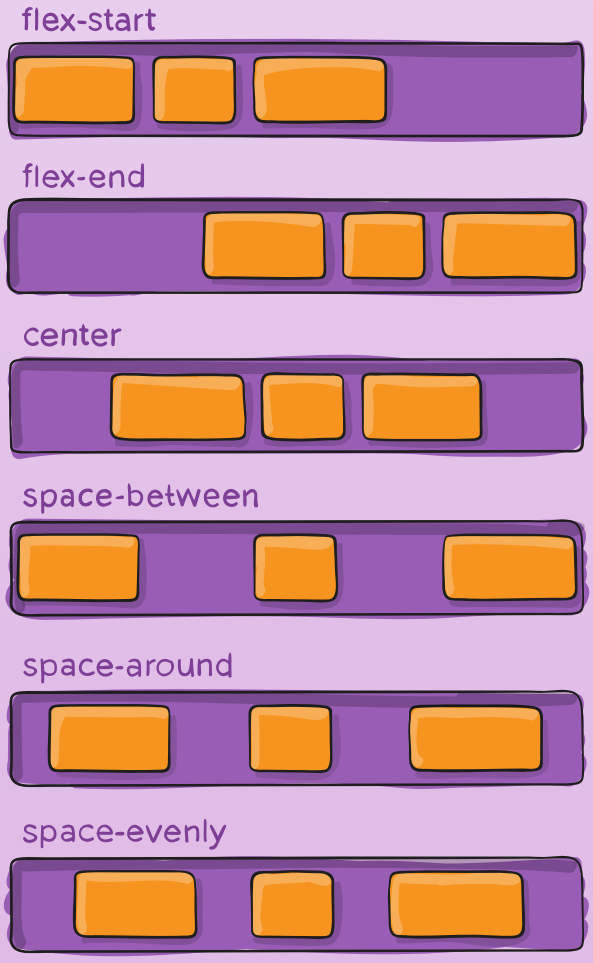
There are some visual demos of `flex-wrap` here (https://css-tricks.com/almanac/properties/f/flex-wrap/) .

(#flex-flow) flex-flow

This is a shorthand for the `flex-direction` and `flex-wrap` properties, which together define the flex container's main and cross axes. The default value is `row nowrap`.

```css
.container {
  flex-flow: column wrap;
}
```

(#justify-content) justify-content



This defines the alignment along the main axis. It helps distribute extra free space leftover when either all the flex items on a line are inflexible, or are flexible but have reached their maximum size. It also exerts some control over the alignment of items when they overflow the line.

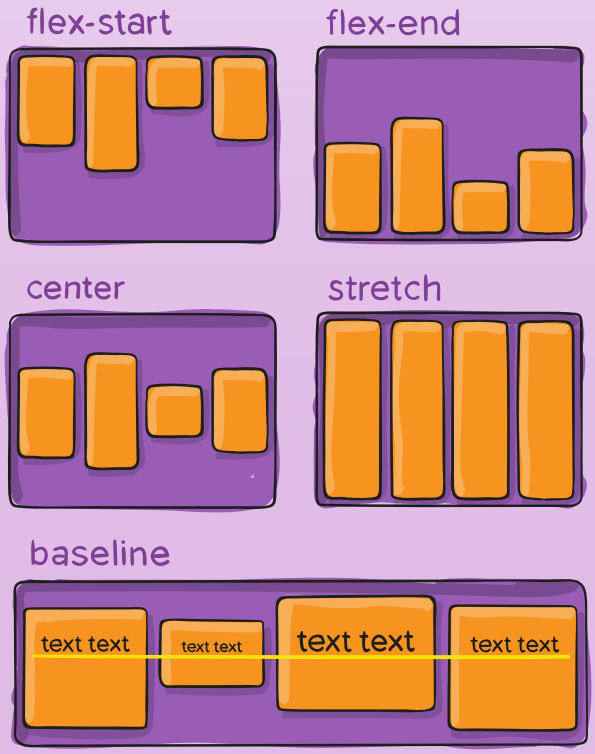
```css
.container {
  justify-content: flex-start | flex-end | center | space-between | space-around | space-evenly | start | end | l
}
```

- `flex-start` (default): items are packed toward the start of the flex-direction.
- `flex-end`: items are packed toward the end of the flex-direction.
- `start`: items are packed toward the start of the `writing-mode` direction.
- `end`: items are packed toward the end of the `writing-mode` direction.
- `left`: items are packed toward left edge of the container, unless that doesn't make sense with the `flex-direction`, then it behaves like `start`.
- `right`: items are packed toward right edge of the container, unless that doesn't make sense with the `flex-direction`, then it behaves like `end`.
- `center`: items are centered along the line
- `space-between`: items are evenly distributed in the line; first item is on the start line, last item on the end line
- `space-around`: items are evenly distributed in the line with equal space around them. Note that visually the spaces aren't equal, since all the items have equal space on both sides. The first item will have one unit of space against the container edge, but two units of space between the next item because that next item has its own spacing that applies.
- `space-evenly`: items are distributed so that the spacing between any two items (and the space to the edges) is equal.

Note that that browser support for these values is nuanced. For example, `space-between` never got support from some versions of Edge, and start/end/left/right aren't in Chrome yet. MDN has detailed charts (https://developer.mozilla.org/en-US/docs/Web/CSS/justify-content) . The safest values are `flex-start`, `flex-end`, and `center`.

There are also two additional keywords you can pair with these values: `safe` and `unsafe`. Using `safe` ensures that however you do this type of positioning, you can't push an element such that it renders off-screen (e.g. off the top) in such a way the content can't be scrolled too (called "data loss").

(#align-items) align-items



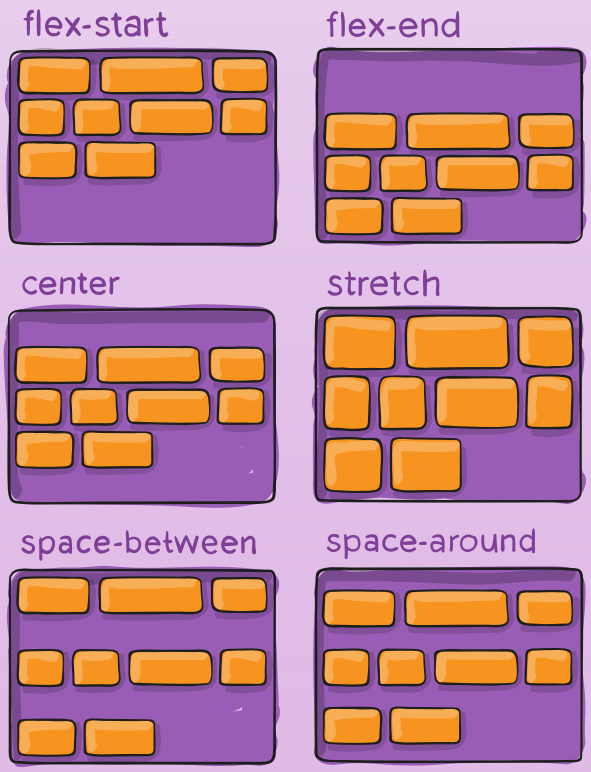
This defines the default behavior for how flex items are laid out along the **cross axis** on the current line. Think of it as the `justify-content` version for the cross-axis (perpendicular to the main-axis).

```css
.container {
  align-items: stretch | flex-start | flex-end | center | baseline | first baseline | last baseline | start | enc
}
```

- `stretch` (default): stretch to fill the container (still respect min-width/max-width)
- `flex-start` / `start` / `self-start`: items are placed at the start of the cross axis. The difference between these is subtle, and is about respecting the `flex-direction` rules or the `writing-mode` rules.
- `flex-end` / `end` / `self-end`: items are placed at the end of the cross axis. The difference again is subtle and is about respecting `flex-direction` rules vs. `writing-mode` rules.
- `center`: items are centered in the cross-axis
- `baseline`: items are aligned such as their baselines align

The `safe` and `unsafe` modifier keywords can be used in conjunction with all the rest of these keywords (although note browser support (https://developer.mozilla.org/en-US/docs/Web/CSS/align-items)), and deal with helping you prevent aligning elements such that the content becomes inaccessible.

(#align-content) align-content



This aligns a flex container's lines within when there is extra space in the cross-axis, similar to how `justify-content` aligns individual items within the main-axis.

> ***Hey!*** **Note:** This property only takes effect on multi-line flexible containers, where `flex-wrap` is set to either `wrap` or `wrap-reverse`). A single-line flexible container (i.e. where `flex-wrap` is set to its default value, `no-wrap`) will not reflect `align-content`.

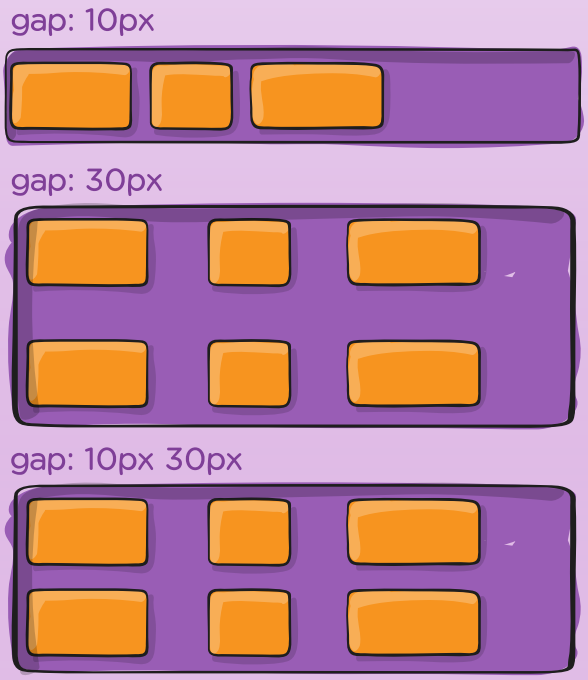
```css
.container {
  align-content: flex-start | flex-end | center | space-between | space-around | space-evenly | stretch | start
}
```

- `normal` (default): items are packed in their default position as if no value was set.
- `flex-start` / `start`: items packed to the start of the container. The (more supported) `flex-start` honors the `flex-direction` while `start` honors the `writing-mode` direction.
- `flex-end` / `end`: items packed to the end of the container. The (more support) `flex-end` honors the `flex-direction` while end honors the `writing-mode` direction.
- `center`: items centered in the container
- `space-between`: items evenly distributed; the first line is at the start of the container while the last one is at the end
- `space-around`: items evenly distributed with equal space around each line
- `space-evenly`: items are evenly distributed with equal space around them

- `stretch`: lines stretch to take up the remaining space

The `safe` and `unsafe` modifier keywords can be used in conjunction with all the rest of these keywords (although note browser support (https://developer.mozilla.org/en-US/docs/Web/CSS/align-items)), and deal with helping you prevent aligning elements such that the content becomes inaccessible.

(#gap-row-gap-column-gap) gap, row-gap, column-gap

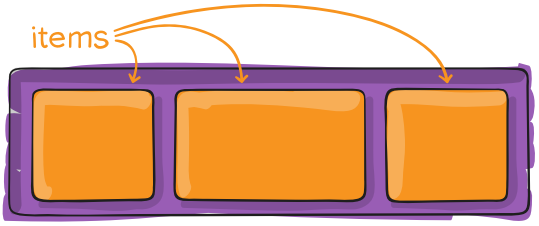


The `gap` property (https://css-tricks.com/almanac/properties/g/gap/) explicitly controls the space between flex items. It applies that spacing *only between items* not on the outer edges.

```css
.container {
  display: flex;
  ...
  gap: 10px;
  gap: 10px 20px; /* row-gap column gap */
  row-gap: 10px;
  column-gap: 20px;
}
```

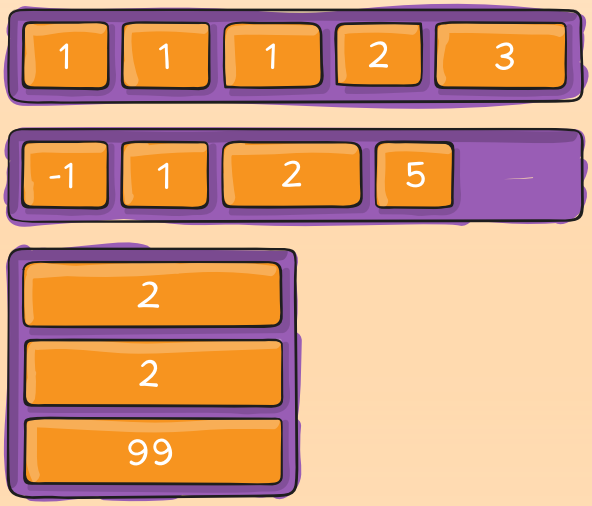
The behavior could be thought of as a *minimum* gutter, as if the gutter is bigger somehow (because of something like `justify-content: space-between;`) then the gap will only take effect if that space would end up smaller.

It is not exclusively for flexbox, `gap` works in grid and multi-column layout as well.



(#properties-for-the-childrenflex-items) Properties for the Children (flex items)

(#order) order

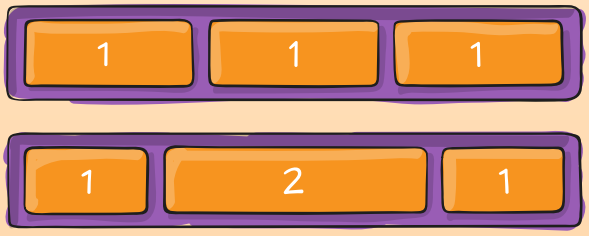


By default, flex items are laid out in the source order. However, the `order` property controls the order in which they appear in the flex container.

```css
.item {
  order: 5; /* default is 0 */
}
```

Items with the same `order` revert to source order.

(#flex-grow) flex-grow



This defines the ability for a flex item to grow if necessary. It accepts a unitless value that serves as a proportion. It dictates what amount of the available space inside the flex container the item should take up.

If all items have `flex-grow` set to 1, the remaining space in the container will be distributed equally to all children. If one of the children has a value of 2, the remaining space would take up twice as much space as the others (or it will try to, at least).

```css
.item {
  flex-grow: 4; /* default 0 */
}
```

Negative numbers are invalid.

(#flex-shrink) flex-shrink

This defines the ability for a flex item to shrink if necessary.

```css
.item {
  flex-shrink: 3; /* default 1 */
}
```

Negative numbers are invalid.

(#flex-basis) flex-basis

This defines the default size of an element before the remaining space is distributed. It can be a length (e.g. 20%, 5rem, etc.) or a keyword. The `auto` keyword means "look at my width or height property" (which was temporarily done by the `main-size` keyword until deprecated). The `content` keyword means "size it based on the item's content" – this keyword isn't well supported yet, so it's hard to test and harder to know what its brethren `max-content`, `min-content`, and `fit-content` do.

```css
.item {
  flex-basis:  | auto; /* default auto */
}
```

If set to `0`, the extra space around content isn't factored in. If set to `auto`, the extra space is distributed based on its `flex-grow` value. See this graphic. (https://www.w3.org/TR/css3-flexbox/images/rel-vs-abs-flex.svg)

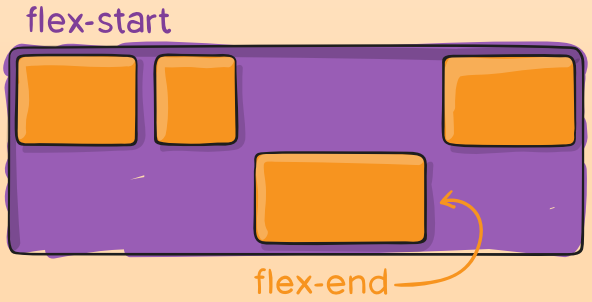
(#flex) flex

This is the shorthand for `flex-grow`, `flex-shrink` and `flex-basis` combined. The second and third parameters (`flex-shrink` and `flex-basis`) are optional. The default is `0 1 auto`, but if you set it with a single number value, it's like `1 0`.

```css
.item {
  flex: none | [ <'flex-grow'> <'flex-shrink'>? || <'flex-basis'> ]
}
```

It is recommended that you use this shorthand property rather than set the individual properties. The shorthand sets the other values intelligently.

(#align-self) align-self



This allows the default alignment (or the one specified by `align-items`) to be overridden for individual flex items.

Please see the `align-items` explanation to understand the available values.

```css
.item {
  align-self: auto | flex-start | flex-end | center | baseline | stretch;
}
```

Note that `float`, `clear` and `vertical-align` have no effect on a flex item.

(#prefixing-flexbox) Prefixing Flexbox

Flexbox requires some vendor prefixing to support the most browsers possible. It doesn't just include prepending properties with the vendor prefix, but there are actually entirely different property and value names. This is because the Flexbox spec has changed over time, creating an "old", "tweener", and "new" (https://css-tricks.com/old-flexbox-and-new-flexbox/) versions.

Perhaps the best way to handle this is to write in the new (and final) syntax and run your CSS through Autoprefixer (https://css-tricks.com/autoprefixer/) , which handles the fallbacks very well.

Alternatively, here's a Sass `@mixin` to help with some of the prefixing, which also gives you an idea of what kind of things need to be done:

```scss
@mixin flexbox() {
  display: -webkit-box;
  display: -moz-box;
  display: -ms-flexbox;
  display: -webkit-flex;
  display: flex;
}

@mixin flex($values) {
  -webkit-box-flex: $values;
  -moz-box-flex:  $values;
  -webkit-flex:  $values;
  -ms-flex:  $values;
  flex:  $values;
}

@mixin order($val) {
  -webkit-box-ordinal-group: $val;
  -moz-box-ordinal-group: $val;
  -ms-flex-order: $val;
  -webkit-order: $val;
  order: $val;
}

.wrapper {
  @include flexbox();
}

.item {
  @include flex(1 200px);
  @include order(2);
}
```

(#examples) Examples

Let's start with a very very simple example, solving an almost daily problem: perfect centering. It couldn't be any simpler if you use flexbox.

```css
.parent {
  display: flex;
  height: 300px; /* Or whatever */
}

.child {
  width: 100px;  /* Or whatever */
  height: 100px; /* Or whatever */
  margin: auto;  /* Magic! */
}
```

This relies on the fact a margin set to `auto` in a flex container absorb extra space. So setting a margin of `auto` will make the item perfectly centered in both axes.

Now let's use some more properties. Consider a list of 6 items, all with fixed dimensions, but can be auto-sized. We want them to be evenly distributed on the horizontal axis so that when we resize the browser, everything scales nicely, and without media queries.

```css
.flex-container {
  /* We first create a flex layout context */
  display: flex;

  /* Then we define the flow direction
     and if we allow the items to wrap
   * Remember this is the same as:
   * flex-direction: row;
   * flex-wrap: wrap;
   */
  flex-flow: row wrap;

  /* Then we define how is distributed the remaining space */
  justify-content: space-around;
}
```

Done. Everything else is just some styling concern. Below is a pen featuring this example. Be sure to go to CodePen and try resizing your windows to see what happens.

Embedded Pen Here

Let's try something else. Imagine we have a right-aligned navigation element on the very top of our website, but we want it to be centered on medium-sized screens and single-columned on small devices. Easy enough.

```css
/* Large */
.navigation {
  display: flex;
```

```
  flex-flow: row wrap;
  /* This aligns items to the end line on main-axis */
  justify-content: flex-end;
}

/* Medium screens */
@media all and (max-width: 800px) {
  .navigation {
    /* When on medium sized screens, we center it by evenly distributing empty space around items */
    justify-content: space-around;
  }
}

/* Small screens */
@media all and (max-width: 500px) {
  .navigation {
    /* On small screens, we are no longer using row direction but column */
    flex-direction: column;
  }
}
```

Embedded Pen Here

Let’s try something even better by playing with flex items flexibility! What about a mobile-first 3-columns layout with full-width header and footer. And independent from source order.

CSS

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

/* We tell all items to be 100% width, via flex-basis */
.wrapper > * {
  flex: 1 100%;
}

/* We rely on source order for mobile-first approach
 * in this case:
 * 1. header
 * 2. article
 * 3. aside 1
 * 4. aside 2
 * 5. footer
 */

/* Medium screens */
@media all and (min-width: 600px) {
  /* We tell both sidebars to share a row */
  .aside { flex: 1 auto; }
}

/* Large screens */
@media all and (min-width: 800px) {
```

```
    /* We invert order of first sidebar and main
     * And tell the main element to take twice as much width as the other two sidebars
     */
    .main { flex: 2 0px; }
    .aside-1 { order: 1; }
    .main    { order: 2; }
    .aside-2 { order: 3; }
    .footer  { order: 4; }
  }
```

Embedded Pen Here

(#flexbox-tricks) Flexbox Tricks

(#browser-support) Browser support

(#bugs) Bugs

(#related-properties) Related properties

(#more-information) More information