CSS Flexbox Ultimate Guide

November 22, 2021

CSS

Laying out elements in CSS is something that used to be pretty difficult to do. We were forced to use annoying hacks like floating elements with clearfixes, but then flexbox was created. Flexbox revolutionized CSS layouts and is by far one of the most defining features of CSS to come out in the last 10 years. In this article I will teach you exactly what flexbox is, how you can use it, and all the advanced nuances you need to understand.

If you prefer to learn visually, check out the video version of this article.

What Is Flexbox?

Flexbox is a way to layout elements in CSS and is broken into a two main components. The flexbox container and flexbox items. The flexbox container is the parent element that contains all the flexbox items as its children. The flexbox container is where you define all your properties about the flexbox layout and then on the individual items you can make additional tweaks. Let's take a look at how we would get started with a simple flexbox example. If we have an element with children inside of it all we need to do is set the display property of the parent element to `flex` and we will have a flex container.

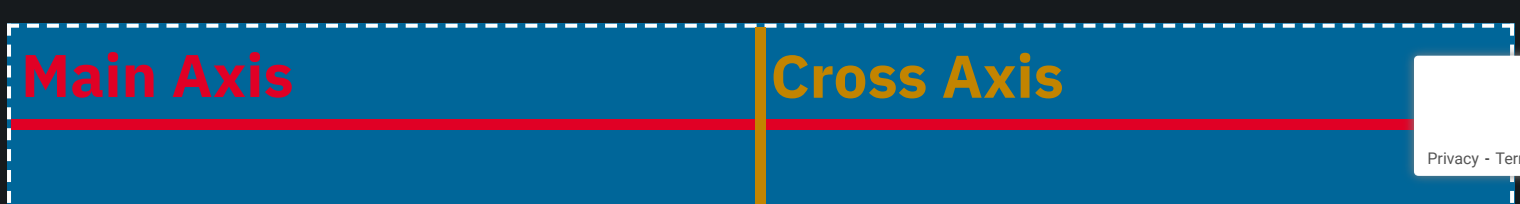
```
display: flex
```

As you can see by default when we specify a display of flex on an element we are setting that element as the flex container. All the direct children in that element are then considered flex items. By default if you specify no other properties the flex items will lay themselves out on one line from left to right taking up only as much space as they need. Items will also automatically shrink down to fit within the flex container if they would normally overflow.

Now this is the absolute basics of flexbox and on its own isn't really useful so next we need to talk about how to handle the layout of items in the container.

Flexbox Layout

Flexbox was the first CSS layout method that worked in a completely different way than normal CSS. Instead of worrying about block/inline elements flexbox worries about a main axis and a cross axis.



By default the main axis (shown in green) goes horizontally across the container and the cross axis goes vertically. This means that any layout method that deals with the main axis will layout elements horizontally while the cross axis will layout elements vertically. Let's look at how we can layout elements along the main axis first.

`justify-content`

For all these examples we will assume that the flex-items have a width of 20%.

```
.flex-item {
  width: 20%;
}
```

`flex-start` (default)

Places all items at the start of the main axis which is the left side of the axis by default. This is also the default behavior of `justify-content`.

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

`flex-end`

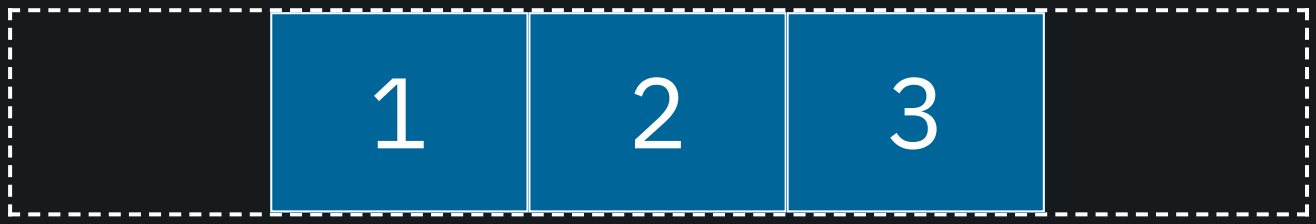
Places all items at the end of the main axis which is the right side of the axis by default.

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

`center`

Places all items in the center of the main axis. This is one of the easiest ways to center elements in CSS.

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



`space-between`

This takes all the extra space inside the container and evenly spreads it between each element to space them as far apart as possible from one another while filling the full container.

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

`space-around`

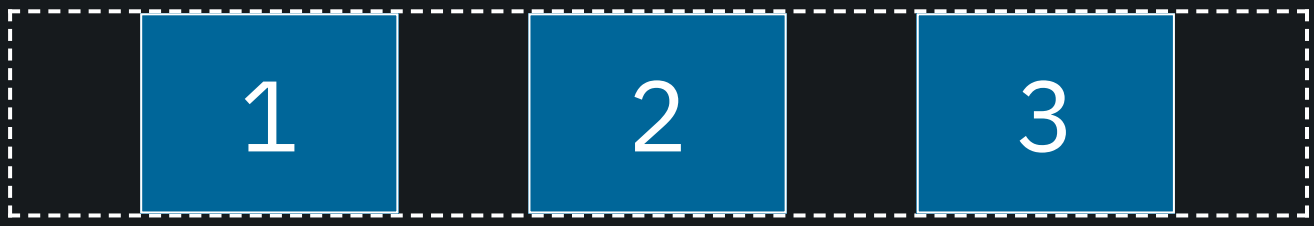
This is very similar to `space-between`, but it also adds space between the outside of the container and the first/last element. The amount of space between the outside of the container and the first/last element is exactly half the amount of space between elements.

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

`space-evenly`

This is very similar to `space-around`, but the space between the outside of the container and the first/last element is the same as the space between elements instead of half the size.

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



With all of these properties you can easily lay elements out along the main axis to fit any need. Now let's talk about how to lay out element on the cross axis.

`align-items`

For all these examples we will assume that the flex-items all have a width of 20%, but that the elements are all different heights.

```
.flex-item {
  width: 20%;
}

.flex-item:nth-child(1) {
  height: 75px;
}

.flex-item:nth-child(2) {
  height: 100px;
}

.flex-item:nth-child(3) {
  height: 125px;
}
```

`stretch` (default)

This will stretch all items to fill the full height of the cross axis unless they have a specific height set. In our example I set the height of the first child to `initial` which is essentially the same as if we had never set a height on the first child. By default when a div has no height it will just be the height of the content inside of it, but as you can see below the first child fills the full height of the container since it is stretching to fill the full height. The second element is not stretching, though, since we set a specific height of 100px on it. This is the default behavior of `align-items`.

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

.flex-item:nth-child(1) {
  /* This is the same as if we had not set a height */
  height: initial;
}
```

`flex-start`

This works the same as `flex-start` for `justify-content`, but will start at the top of the cross axis by default.

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

`flex-end`

This works the same as `flex-end` for `justify-content`, but will start at the bottom of the cross axis by default.

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

`center`

This works the same as `center` for `justify-content`, but will center based on the cross axis.

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

Now this covers all the ways you can layout elements along the main and cross axis, but there is one more important thing you need to know about flexbox axes. They can actually be swapped. There is a property called `flex-direction` which determines the orientation of the main and cross axis.

`flex-direction`

This property allows us to determine which direction each axis corresponds to as well as where the axis start.

`row` (default)

The default direction is row. This means the main axis is horizontal while the cross axis is vertical. This also means the main axis starts on the left while the cross axis starts at the top.

```
.flex-container {
  display: flex;
  flex-direction: row;
  justify-content: flex-start;
  align-items: flex-start;
}
```

`row-reverse`

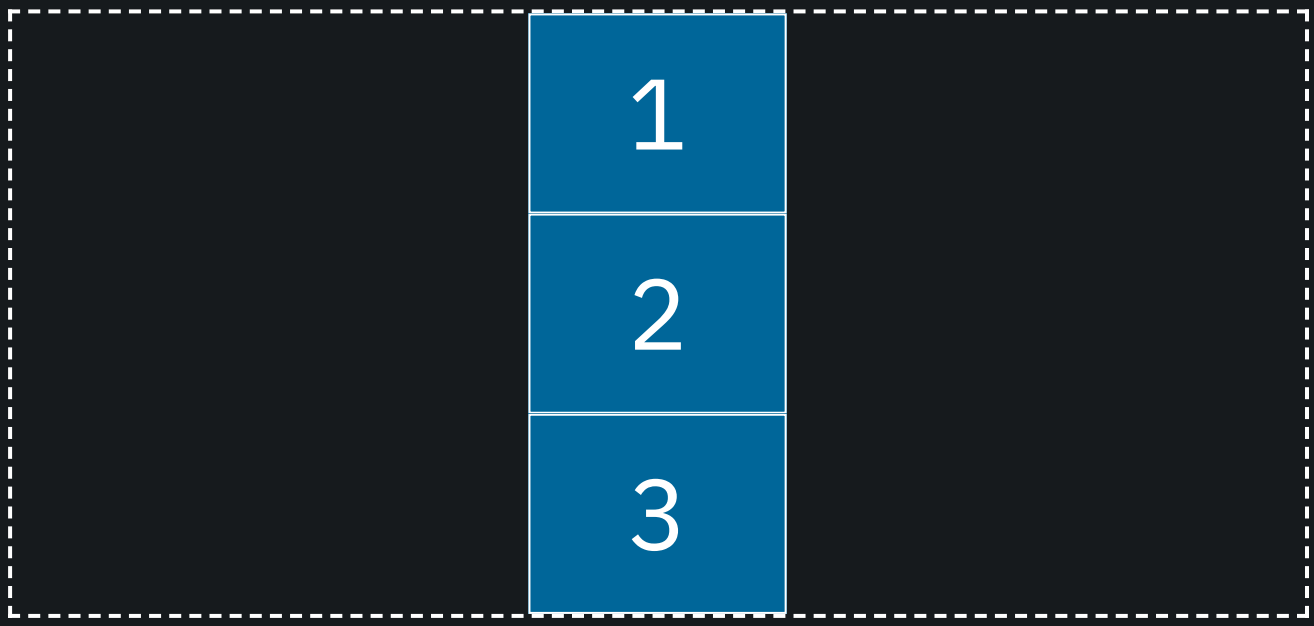
Similar to `row` we have `row-reverse`. This direction does **not** swap the main/cross axis, but it does swap where the main axis starts. The main axis now starts on the right while the cross axis does **not** change and still starts at the top. You will see below that our items start on the right side of the container and are ordered right to left since we are using the reverse ordering.

```
.flex-container {
  display: flex;
  flex-direction: row-reverse;
  justify-content: flex-start;
  align-items: flex-start;
}
```

`column`

The `column` direction completely swaps our axes so now the main axis is vertical and the cross axis is horizontal. This means that if you use `justify-content` you will be laying out elements in the vertical direction and `align-items` will work in the horizontal direction.

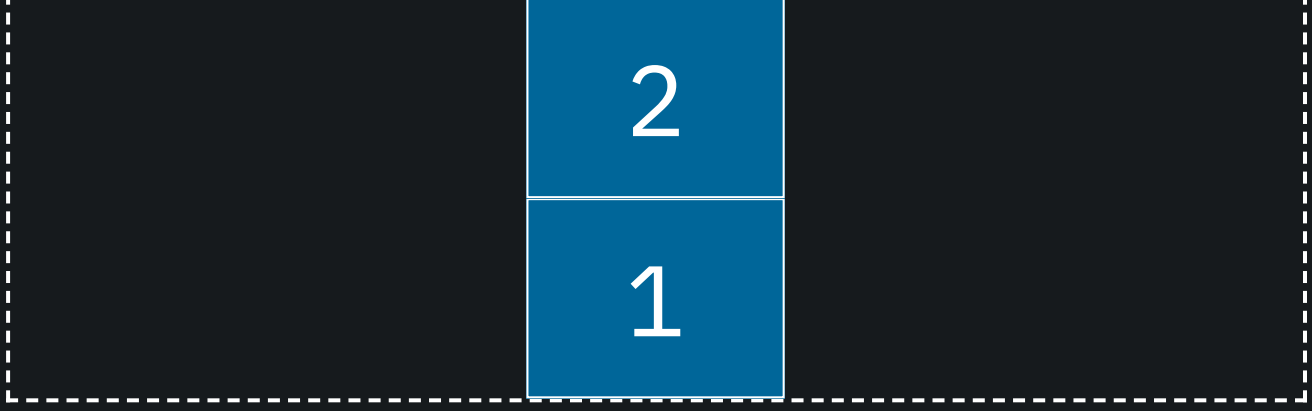
```
.flex-container {
  display: flex;
  flex-direction: column;
  justify-content: flex-start;
  align-items: center;
}
```



`column-reverse`

The `column-reverse` direction is essentially the same as `column`, but it reverses the direction of the main axis so now the items start at the bottom of the container.

```
.flex-container {
  display: flex;
  flex-direction: column-reverse;
  justify-content: flex-start;
  align-items: center;
}
```



Now that is a lot of stuff we just covered related to layout, but really the main things you need to understand are `justify-content`, `align-items`, `flex-direction`, and how each of those effect the main/cross axis.

Flex Item Layout

So far everything we have covered has had to deal with the layout of the entire flex container. We can actually take this a step further, though, and have specific layouts for each item in the flex container. This is done via the `align-self` property.

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

.flex-item:nth-child(1) {
  align-self: flex-end;
}
```

As you can see from the above example we set the `align-self` property of the first child to `flex-end` and it is now aligned at the bottom of our cross axis even though the flex container as a whole has an `align-items` of `flex-start`.

This `align-self` property can be used with any of the `align-items` values to place each item in the container exactly where you want them.

One important thing to note, though, is that there is no way to do `justify-self` since all justification is taken care of by the parent only.

Sizing Flex Items

Now we come to what is probably the most confusing part of flexbox which is sizing the individual items, but I promise you I will make it as easy as possible to understand.

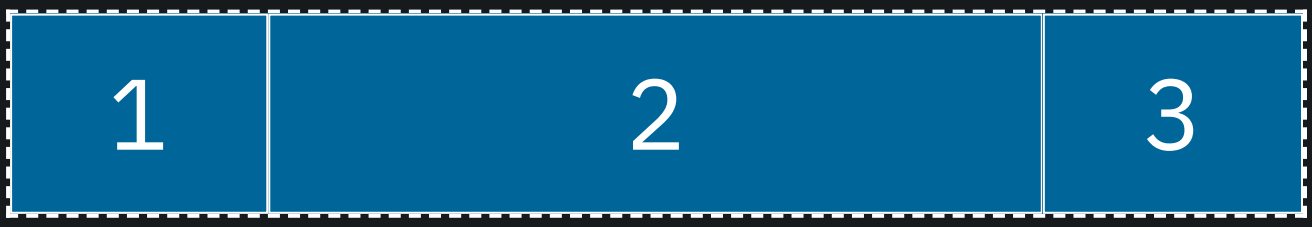
The real power of flexbox is not in its layout properties (even though, those are amazing), but in its ability to resize items based on the size of other elements on your page. This is done via 3 different properties `flex-grow`, `flex-shrink`, and `flex-basis`. First I want to talk about `flex-grow` since it is the most common property you will use.

`flex-grow`

The `flex-grow` property is a property you define on a flex item and it tells the item how much of the extra space that item is allowed to take to fill its container. By default this property is set to 0 which means the item does not get any extra space. Let's first look at a flex container where none of the items have `flex-grow` set.

As you can see each item is taking up only its width and the rest of the space in the container is unfilled. This is the default behavior of flexbox, but if you want one or more items to fill the remaining space you need `flex-grow`.

```
.flex-item:nth-child(2) {
  flex-grow: 1;
}
```



By setting a `flex-grow` of 1 we are telling the second element that it should get 1 part of the extra space and since no other items have a flex grow that 1 part of extra space is the entirety of the extra space.

```
.flex-item:nth-child(1),
.flex-item:nth-child(2) {
  flex-grow: 1;
}
```

In this example we set both the first and second element to a `flex-grow` of 1 so now each of those element will receive 1 part of the remaining space. To determine how much space that is we just add up all the `flex-grow` numbers for each item in the container (1 + 1 = 2) and then divide the `fl`

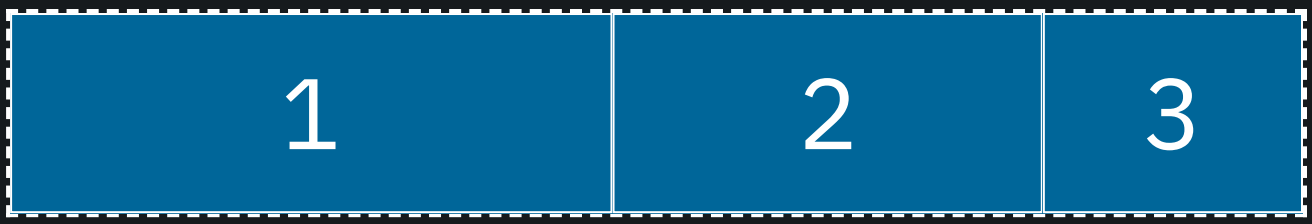
`grow` of each item by that number. Since our `flex-grow` for each item is 1 each item will get 1/2 of the remaining space added to it.

We can make more complex layouts, though, by giving some elements more or less of the available space.

```
.flex-item:nth-child(1) {
  flex-grow: 2;
}

.flex-item:nth-child(2) {
  flex-grow: 1;
}
```

In this example we are saying the first element should get 2 parts of the remaining space while the second element only gets 1 part. When we do the math we will see that the first element gets 2/3 of the remaining space while the second element only gets 1/3 the remaining space.



At first glance you may think this code is saying that the first element should be twice the size of the second element but flex grow only cares about the remaining space after all elements are added to the container. Since by default our 3 elements take up 60% of the container size the remaining space to divide between the elements is only 40% of the container size. We can actually modify how this remaining space is calculated, though, by using `flex-basis`

`flex-basis`

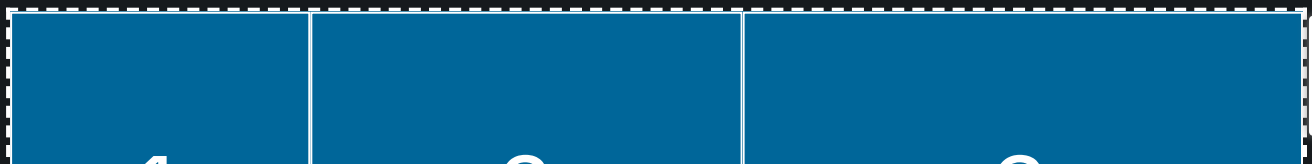
The `flex-basis` property tells our flex container how much space the item is taking up in the container. By default this is set to `auto` which means it just uses the `width` property of the element to calculate this.

```
.flex-item:nth-child(1) {
  flex-basis: 40%;
}
```

In the above code we set the `flex-basis` of the first element to 40% so now instead of taking up 20% of the width (which is what we have defined as the `width` of the element) it will instead take up 40% of the container size. This now means there is less space remaining in the container to divide up with `flex-grow`.

This property is really only useful when you want to change the width of elements in the flex container so they work better with `flex-grow`. For example let's say you want to create 3 equal size elements in your container, but they all have different widths to start with that you cannot change.

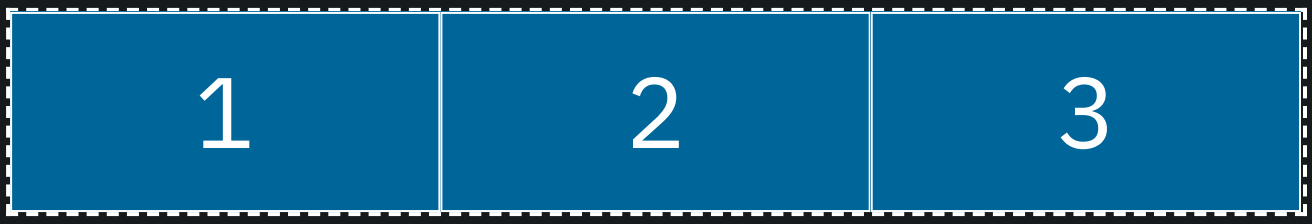

```
.flex-item {
  flex-grow: 1;
}
```



You may think just setting `flex-grow` to 1 would make them all the same size, but since they start out at different widths they will end up at different widths since the amount of extra space added to the elements is exactly the same for each independent of their size.

To get around this issue you need to give all elements the same `flex-basis` so they all start at the same size and grow at the same rate. Generally you will use 0 for the `flex-basis` since it will ensure all elements are always the same size no matter how small/large they start.

```
.flex-item {
  flex-grow: 1;
  flex-basis: 0;
}
```

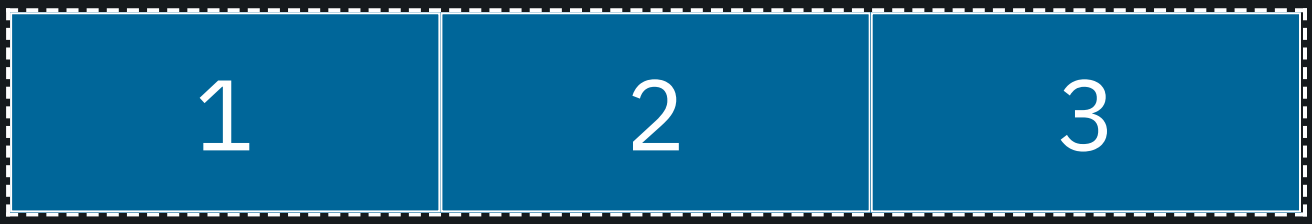


Now the last sizing property we have to talk about is `flex-shrink` which we have actually already seen in action.

`flex-shrink`

Way back at the start of this article I mentioned how if the flex items in a container overflow the container they will automatically shrink down to fill the correct size. This is because by default `flex-shrink` is set to 1 on all flex items.

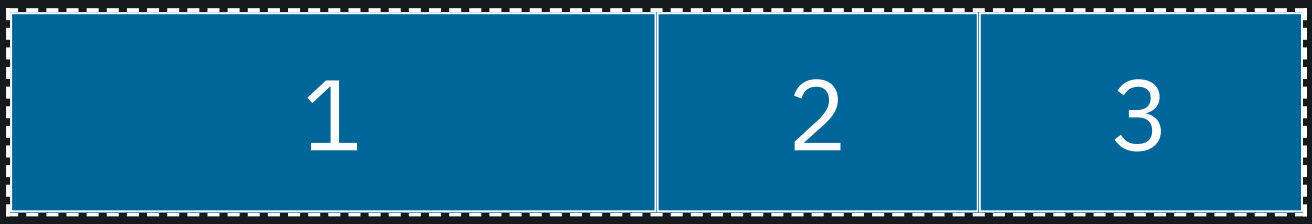
```
.flex-item {
  width: 50%;
}
```



Even though each element should be 50% of the container they shrink down evenly so that each element is only 33% of the container size. If we wanted to prevent one of the children from shrinking we could set the `flex-shrink` to 0.

```
.flex-item {
  width: 50%;
}

.flex-item:nth-child(1) {
  flex-shrink: 0;
}
```



As you can see the first element stays 50% of the container size and does not shrink while the other two elements shrink down to ensure all items can fit in the container.

We can also make it so that some items shrink more than other.

```css
.flex-item {
  width: 50%;
}

.flex-item:nth-child(1) {
  flex-shrink: 2;
}
```

By setting `flex-shrink` to 2 we are saying that the first element should lose 2 parts of the overflown space while the other two elements each only lose 1 part since they are set to a `flex-shrink` of 1 by default. This works exactly the same as `flex-grow` when it comes to proportions, but `flex-shrink` deals with the overflown space outside the container while `flex-grow` deals with the space left over inside the container.

For the most part this is not really a property that you will have to mess with much since you usually only care about growing items and the default of shrinking when overflown is usually what you want.

Also, an important thing to note is that `flex-shrink` is pretty smart and will ensure that if you have one really large item and one really small item that they will shrink in a way that the large item shrinks more so that the small items doesn't shrink so small that it disappears.

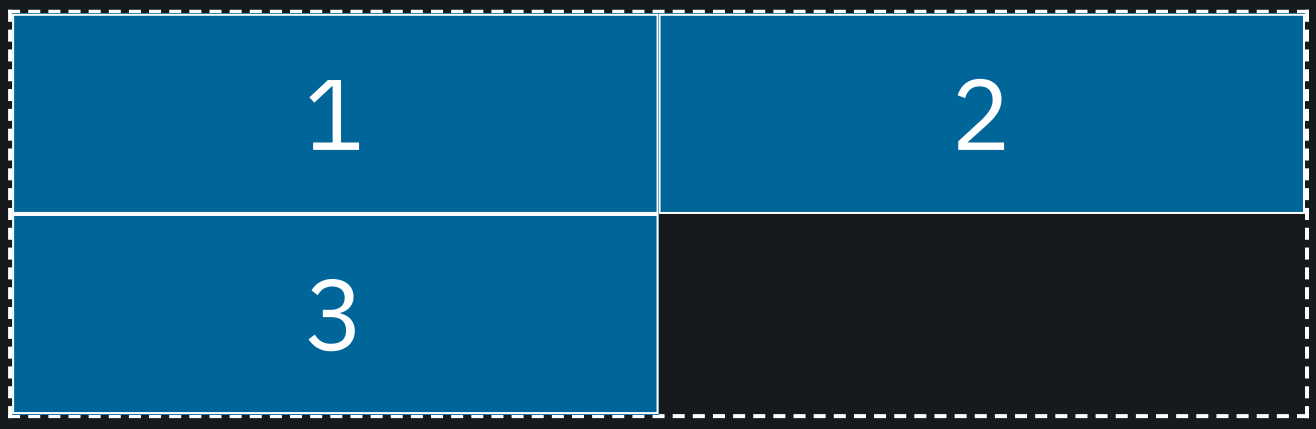
Now that is all you need to know about sizing flex items, but what happens if you want to ensure items don't shrink and that they can wrap to a new line as needed.

Flex Wrapping

When you are working in flexbox you are usually working with just one line of items, but sometimes in rare occasions you may want to handle wrapping when you have more items than you can fit on one line.

```
.flex-container {
  flex-wrap: wrap;
}

.flex-item {
  width: 50%;
}
```



Since our items have a width of 50% normally they would shrink to ensure they all fit on one line, but we set the `flex-wrap` property to `wrap` which means that if an item would overflow the container it is instead put on a new line. This is fine if you are using wrap to ensure that even if your list of items grows to be too large they still will look good, but if you are using wrapping to create a two dimensional layout you are using flexbox wrong. Instead you should look towards using CSS grid. You can check out my CSS grid tutorial video for more information on CSS grid.

If you want to stop a flex container from wrapping you need to set the wrap property to `nowrap`.

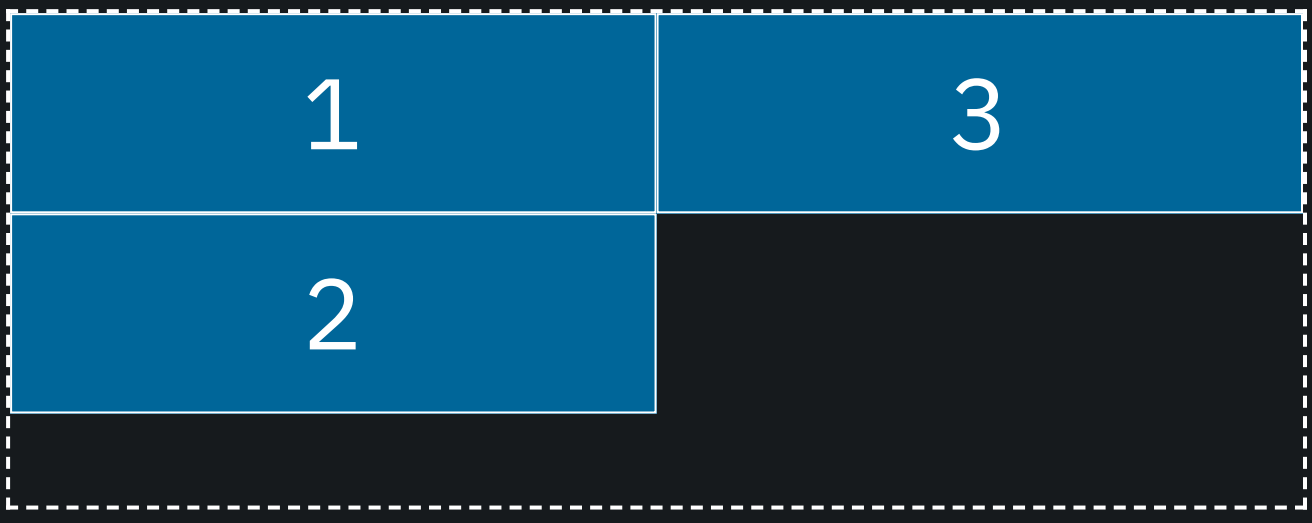
```
.flex-container {
  flex-wrap: nowrap;
}
```

Advanced Wrapping Layout

If you use `flex-wrap` in a column layout then the items will wrap onto new columns instead of rows.

```
.flex-container {
  flex-direction: column;
  height: 250px;
  flex-wrap: wrap;
}

.flex-item {
  height: 100px;
}
```



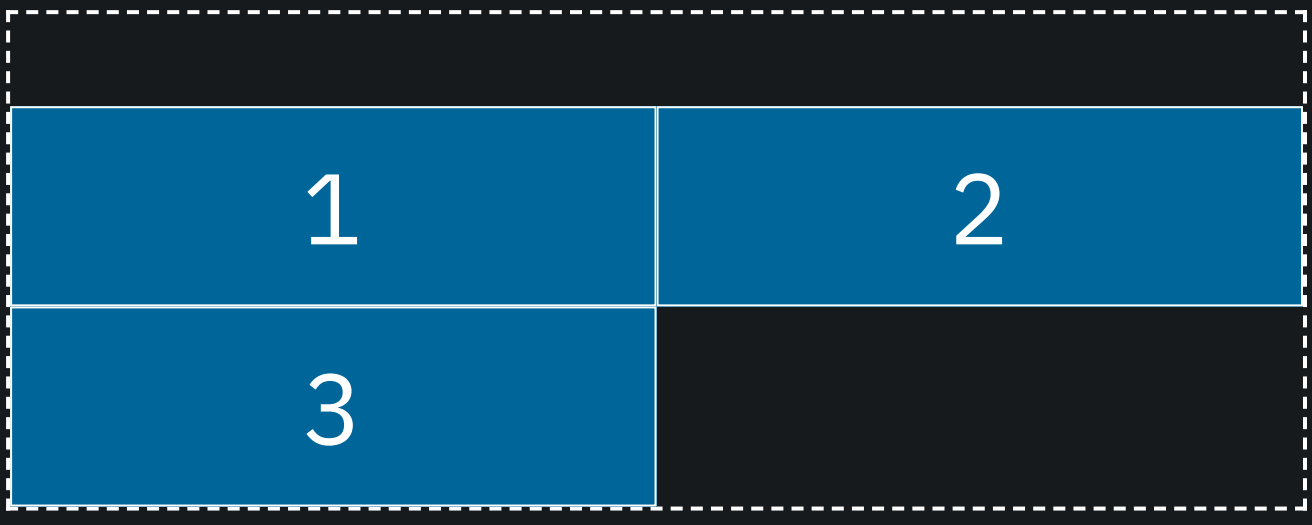
Another important thing to know is that now that there are multiple rows/columns you can use `align-content` to layout how the different rows/columns interact.

In a normal flexbox layout you can use `justify-content` to space the items on the main axis and `align-content` is essentially the same thing, but for spacing out elements on the cross axis. This property will only matter when `flex-wrap` is set to wrap and the content is wrapping.

For example in a normal flexbox layout the `align-content` property is set to `normal` which means that no specific alignment is set. If we want we can use all the fancy alignments available to `justify-content` within `align-content`.

```
.flex-container {
  height: 250px;
  flex-wrap: wrap;
  align-content: flex-end;
}

.flex-item {
  width: 50%;
  height: 100px;
}
```



As you can see all the items are positioned at the bottom of the container which is the end of the cross axis. If we were to use a column layout instead then `align-content` would position the elements horizontally.

```css
.flex-container {
  flex-direction: column;
  height: 250px;
  flex-wrap: wrap;
  align-content: center;
}

.flex-item {
  height: 100px;
}
```

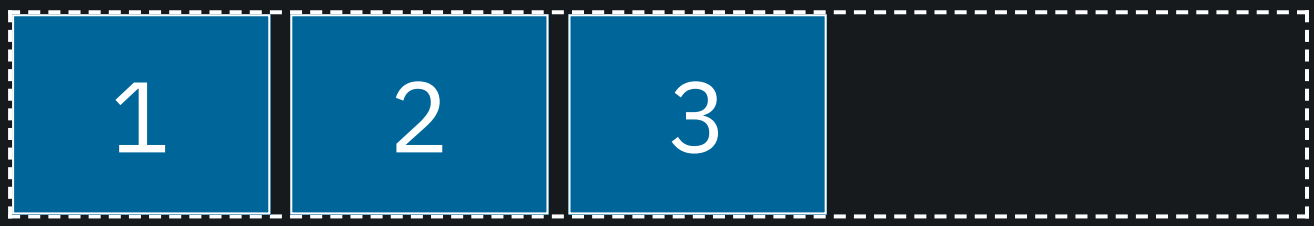
Advanced Flexbox Properties

Now this covers all the basic and even intermediate properties for flexbox so now I want to cover a few advanced/niches flexbox properties.

Adding Gaps

One thing that we haven't covered so far is the ability to add gaps between elements in a flexbox container. You could try to do this with margins/padding, but the easier way is to use the `gap` property. This property will add spac between each item based on the value we pass to `gap`.

```css
.flex-container {
  gap: 10px;
}
```



As you can see each of our items above has a gap of 10px between them. This gap also works for multiline flex containers too.

```css
.flex-container {
  flex-direction: column;
  height: 250px;
  flex-wrap: wrap;
  align-content: center;
  gap: 10px;
}

.flex-item {
  height: 100px;
}
```

As you can see each of our rows/columns has a 10px gap.

We can also go a step further and define only a row gap or only a column gap.

```
.flex-container {
  flex-direction: column;
  height: 250px;
  flex-wrap: wrap;
  align-content: center;
  row-gap: 10px;
}

.flex-item {
  height: 100px;
}
```


```
.flex-container {
  flex-direction: column;
  height: 250px;
  flex-wrap: wrap;
  align-content: center;
  column-gap: 10px;
}

.flex-item {
  height: 100px;
}
```

Ordering Elements

By default we can order elements in normal order or reverse order by using `flex-direction`, but we can also order individual elements with the `order` property.

```
.flex-item:nth-child(1) {
  order: 2
}

.flex-item:nth-child(2) {
  order: 1
}

.flex-item:nth-child(3) {
  order: 1
}
```

As you can see by specifying the `order` property of our elements we can determine the exact order of them. By default each element has an order of 0 and since flexbox will layout out all elements with the same order based on their order in the HTML you will notice that if you don't define any order all elements will stay in the same order they are in the HTML.

In our example we specified an order of 2 for our first item and then an order of 1 for our other two items. When flexbox lays out the items it starts with the lowest order first, which in our case is 1. Since our second and third element both have the same order they will be ordered based on their HTML order so the second element will come first and the third element will come second. Finally our first element has an order of 2 which is large than 1 so it will be put as our last element.

Now this may seem like a cool trick that allows you to really change how your page works, but I recommend never using the `order` property. The reason for this is because it does not work well with screen readers since screen readers always read based on the HTML order. So for example in our above scenario a person looking at the screen will see the numbers in the order 2, 3, 1 while a screen reader will see them in the order 1, 2, 3 since that is the order they appear in the HTML.

Flex Shorthand

We already talked about the 3 main properties used to size a flex item, `flex-grow`, `flex-shrink`, and `flex-basis`. Usually you will only need to define one of these at a time, but if you want to define multiple at once you can use the `flex` shorthand property which defines all 3 properties.

```
.flex-item {
  flex: 1 0 10px;
}
```

The above code is the same as:

```
.flex-item {
  flex-grow: 1;
  flex-shrink: 0;
  flex-basis: 10px;
}
```

The way this property works is the first value is passed to `flex-grow`, the second value is passed to `flex-shrink`, and the third property is passed to `flex-basis`. If you only define the first property then it will just set the `flex-grow` property. CSS is smart enough, though, that if you pass a width value, such as 10px, as the only property to `flex` then it will set just the `flex-basis`.

```
.flex-item {
  flex: 2;
}
/* Same as */
.flex-item {
  flex-grow: 2;
  flex-shrink: 1;
  flex-basis: 0;
}
```

```
.flex-item {
  flex: 10px;
}
/* Same as */
.flex-item {
  flex-grow: 0;
  flex-shrink: 1;
  flex-basis: 10px;
}
```

When using the `flex` shorthand `flex-grow` will default to 0 unless defined, `flex-shrink` will default to 1, and `flex-basis` will default to 0. This is important to note since `flex-basis` is normally `auto`, but if you use the `flex` shorthand it will default `flex-basis` to 0 unless defined.

Conclusion

And finally we come to the end of the ultimate guide to flexbox. I know this was a ton of information to cover, but hopefully at least some of it stuck. I know it is impossible to memorize all this at once, but you can continually come back to this article for quick refreshers on what each property does. I also am planning to create a cheat sheet soon that will cover all flexbox properties/values with nice drawings to go along with them. If that is something you are interested in make sure you sign up for my newsletter at the top of the page to be notified when I finish the cheat sheet.