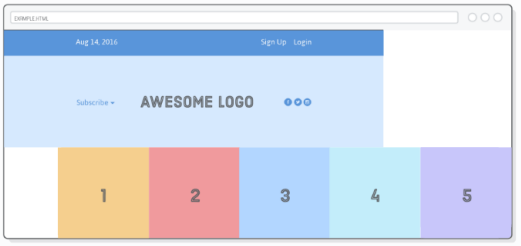
By default, they flow off the edge of the page:



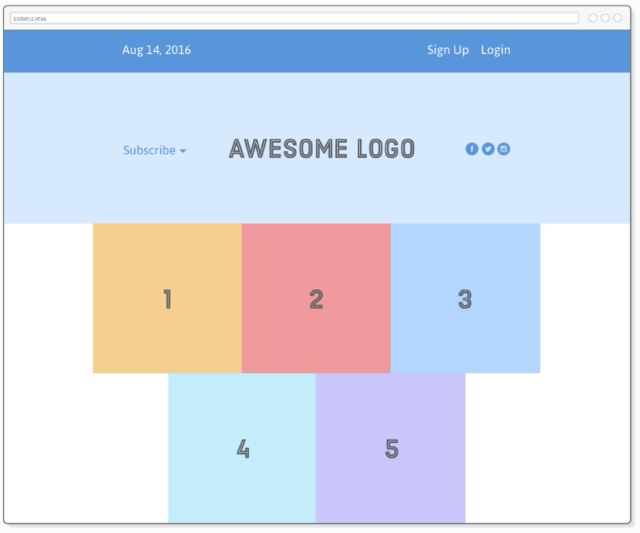
If you're trying to build a hero banner that lets the user horizontally scroll through a bunch of photos, this might be desired behavior, but that's not what we want. Adding the following `flex-wrap` property forces items that don't fit to get bumped down to the next row:

```
.photo-grid {
  /* ... */
  flex-wrap: wrap;
}
```

Now, our flex items behave much like floated boxes, except flexbox gives us more control over how "extra" items are aligned in the final row via the `justify-content` property. For example, the last line is currently left-aligned. Try centering it by updating our `.photo-grid` rule, like so:

```
.photo-grid {
  width: 900px;
  display: flex;
  justify-content: center;    /* Change this */
  flex-wrap: wrap;
}
```

Achieving this with float-based layouts is ridiculously complicated.



FLEX CONTAINER DIRECTION

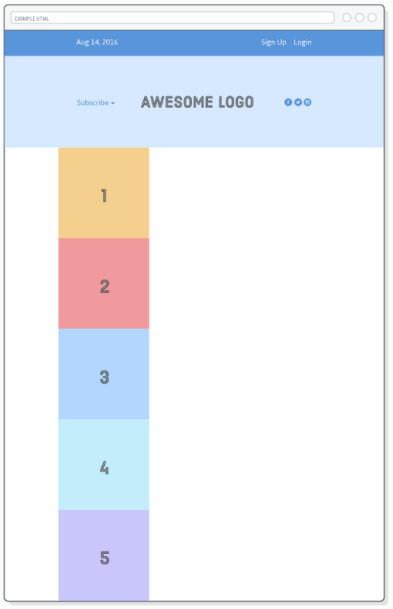
"Direction" refers to whether a container renders its items horizontally or vertically. So far, all the containers we've seen use the default horizontal direction, which means items are drawn one after another in the same row

before popping down to the next column when they run out of space.

One of the most amazing things about flexbox is its ability to transform rows into columns using only a single line of CSS. Try adding the following `flex-direction` declaration to our `.photo-grid` rule:

```
.photo-grid {
  /* ... */
  flex-direction: column;
}
```

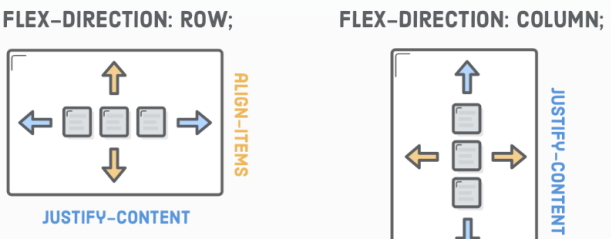
This changes the direction of the container from the default `row` value. Instead of a grid, our page now has a single vertical column:



A key tenant of responsive design is presenting the same HTML markup to both mobile and desktop users. This presents a bit of a problem, as most mobile layouts are a single column, while most desktop layouts stack elements horizontally. You can imagine how useful `flex-direction` is going to become once we start building responsive layouts.

ALIGNMENT CONSIDERATIONS

Notice that the column is hugging the left side of its flex container despite our `justify-content: center;` declaration. When you rotate the direction of a container, you also rotate the direction of the `justify-content` property. It now refers to the container's vertical alignment–not its horizontal alignment.



To horizontally center our column, we need to define an `align-items` property on our `.photo-grid`:

```
.photo-grid {
  /* ... */
  flex-direction: column;
  align-items: center;      /* Add this */
}
```

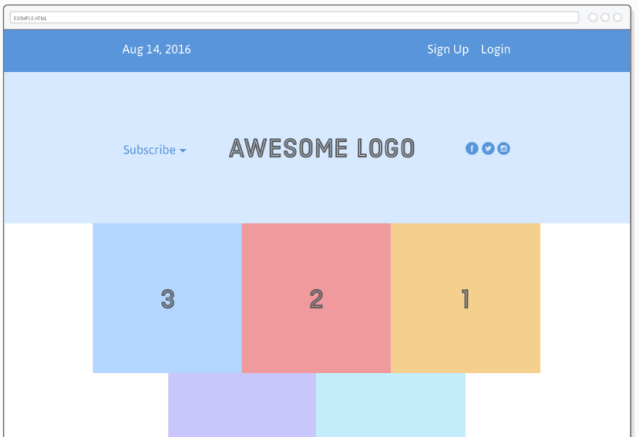
FLEX CONTAINER ORDER

Up until now, there's been a tight correlation between the order of our HTML elements and the way boxes are rendered in a web page. With either floats or the flexbox techniques we've seen so far, the only way we could make a box appear before or after another one is to move around the underlying HTML markup. That's about to change.

The `flex-direction` property also offers you control over the order in which items appear via the `row-reverse` and `column-reverse` properties. To see this in action, let's transform our column back into a grid, but this time around we'll reverse the order of everything:

```
.photo-grid {
  width: 900px;
  display: flex;
  justify-content: center;
  flex-wrap: wrap;
  flex-direction: row-reverse;  /* <--- Really freaking cool! */
  align-items: center;
}
```

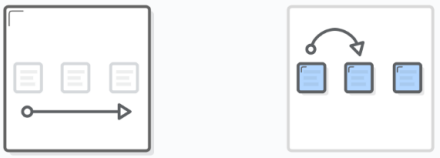
Both rows are now rendered right-to-left instead of left-to-right. But, notice how this only swaps the order on a per-row basis: the first row doesn't start at 5, it starts at 3. This is useful behavior for a lot of common design patterns (`column-reverse` in particular opens up a lot of doors for mobile layouts). We'll learn how to get even more granular in the next section.



Reordering elements from inside a stylesheet is a big deal. Before flexbox, web developers had to resort to JavaScript hacks to accomplish this kind of thing. However, don't abuse your newfound abilities. As we discussed in the very first chapter of this tutorial, you should always separate content from presentation. Changing the order like this is purely presentational—your HTML should still make sense without these styles applied to it.

FLEX ITEM ORDER

This entire chapter has been about positioning flex items *through their parent containers*, but it's also possible to manipulate individual items. The rest of this chapter is going to shift focus away from flex containers onto the items they contain.



FLEX-DIRECTION
(WHOLE CONTAINER)

ORDER
(INDIVIDUAL ITEMS)

Adding an `order` property to a flex item defines its order in the container without affecting surrounding items. Its default value is `0`, and increasing or decreasing it from there moves the item to the right or left, respectively.

This can be used, for example, to swap order of the `.first-item` and `.last-item` elements in our grid. We should also change the `row-reverse` value from the previous section back to `row` because it'll make our edits a little easier to see:

```
.photo-grid {
  /* ... */
  flex-direction: row;  /* Update this */
  align-items: center;
}

.first-item {
  order: 1;
}

.last-item {
  order: -1;
}
```

Unlike setting `row-reverse` and `column-reverse` on a flex container, `order` works across row/column boundaries. The above snippet will switch our first and last items, even though they appear on different rows.

FLEX ITEM ALIGNMENT

We can do the same thing with vertical alignment. What if we want that **Subscribe** link and those social icons to go at the bottom of the header instead of the center? Align them individually! This is where the `align-self` property comes in. Adding this to a flex item overrides the `align-items` value from its container:

```
.social,
.subscribe {
  align-self: flex-end;
  margin-bottom: 20px;
}
```

This should send them to the bottom of the `.header`. Note that margins (padding, too) work just like you'd expect.

You can align elements in other ways using the same values as the `align-items` property, listed below for convenience.

- `center`
- `flex-start` (top)
- `flex-end` (bottom)
- `stretch`
- `baseline`

FLEXIBLE ITEMS

All our examples have revolved around items with fixed- or content-defined-widths. This has let us focus on the positioning aspects of flexbox, but it also means we've been ignoring its eponymous "flexible box" nature. Flex items are *flexible*: they can shrink and stretch to match the width of their containers.

The `flex` property defines the width of individual items in a flex container. Or, more accurately, it allows them to have flexible widths. It works as a weight that tells the flex container how to distribute extra space to each item. For example, an item with a `flex` value of `2` will grow twice as fast as items with the default value of `1`.

First, we need a footer to experiment with. Stick this after the `.photo-grid-container` element:

```
<div class='footer'>
  <div class='footer-item footer-one'></div>
  <div class='footer-item footer-two'></div>
  <div class='footer-item footer-three'></div>
</div>
```

Then, some CSS:

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

.footer-item {
  border: 1px solid #fff;
  background-color: #D6E9FE;
  height: 200px;
  flex: 1;
}
```

That `flex: 1;` line tells the items to stretch to match the width of `.footer`. Since they all have the same weight, they'll stretch equally:

Increasing the weight of one of the items makes it grow faster than the others. For example, we can make the third item grow twice as fast as the other two with the following rule:

```
.footer-three {
  flex: 2;
}
```

Compare this to the `justify-content` property, which distributes extra space *between* items. This is similar, but now we're distributing that space into the items themselves. The result is full control over how flex items fit into their containers.

STATIC ITEM WIDTHS

We can even mix-and-match flexible boxes with fixed-width ones. `flex: initial` falls back to the item's explicit `width` property. This lets us combine static and flexible boxes in complex ways.

We're going to make our footer behave like the above diagram. The center item is flexible, but the ones on either side are always the same size. All we need to do is add the following rule to our stylesheet:

```
.footer-one,
.footer-three {
  background-color: #5995DA;
  flex: initial;
  width: 300px;
}
```

Without that `flex: initial;` line, the `flex: 1;` declaration would be inherited from the `.footer-item` rule, causing the `width` properties to be ignored. `initial` fixes this, and we get a flexible layout that also contains fixed-width items. When you resize the browser window, you'll see that only the middle box in the footer gets resized.

This is a pretty common layout, and not just in footers, either. For instance, many websites have a fixed-width sidebar (or multiple sidebars) and a flexible content block containing the main text of the page. This is basically a taller version of the footer we just created.

FLEX ITEMS AND AUTO-MARGINS

Auto-margins in flexbox are special. They can be used as an alternative to an extra `<div>` when trying to align a group of items to the left/right of a container. Think of auto-margins as a "divider" for flex items in the same container.

Let's take a look by flattening our items in `.menu` so that it matches the following:

```
<div class='menu-container'>
  <div class='menu'>
    <div class='date'>Aug 14, 2016</div>
    <div class='signup'>Sign Up</div>
    <div class='login'>Login</div>
  </div>
</div>
```

Reloading the page should make the items spread out equally through our menu, just like at the beginning of the chapter. We can replicate the desired layout by sticking an auto-margin between the items we want to separate,

like so:

```
.signup {
  margin-left: auto;
}
```

Auto-margins eat up *all* the extra space in a flex container, so instead of distributing items equally, this moves the `.signup` and any following items (`.login`) to the right side of the container. This will give you the exact same layout we had before, but without that extra nested `<div>` to group them. Sometimes, it's nice to keep your HTML flatter.

SUMMARY

Flexbox gave us a ton of amazing new tools for laying out a web page. Compare these techniques to what we were able to do with floats, and it should be pretty clear that flexbox is a cleaner option for laying out modern websites:

- Use `display: flex;` to create a flex container.
- Use `justify-content` to define the horizontal alignment of items.
- Use `align-items` to define the vertical alignment of items.
- Use `flex-direction` if you need columns instead of rows.
- Use the `row-reverse` or `column-reverse` values to flip item order.
- Use `order` to customize the order of individual elements.
- Use `align-self` to vertically align individual items.
- Use `flex` to create flexible boxes that can stretch and shrink.

Remember that these flexbox properties are just a language that lets you tell browsers how to arrange a bunch of HTML elements. The hard part isn't actually writing the HTML and CSS code, it's figuring out, conceptually (on a piece of paper), the behavior of all the necessary boxes to create a given layout.

When a designer hands you a mockup to implement, your first task is to draw a bunch of boxes on it and determine how they're supposed to stack, stretch, and shrink to achieve the desired design. Once you've got that done, it should be pretty easy to code it up using these new flexbox techniques.

The flexbox layout mode should be used for most of your web pages, but there are some things it's not-so-good at, like gently tweaking element positions and preventing them from interacting with the rest of the page. After covering these kinds of advanced positioning techniques in the next chapter, you'll be an HTML and CSS positioning expert.

NEXT CHAPTER >

More tutorials are coming. (Lots more.)
Enter your email above, and we'll let you know when they get here.