1. Style sheets for web development: CSS

1.1 Introduction

Learning objectives, concepts, and skills:

- Describe the key elements of the CSS language
- Define styles in CSS
- Associate CSS styles with predefined elements of HTML documents
- Create classes of elements that allow you to style elements not predefined in HTML
- Describe some properties of the CSS language

1.2 The CSS language

CSS (Cascading Style Sheet) is used to define the styles of web pages. It is used to define the styles of web pages, including their layout, design, variations in display for various devices, and sizes of screens. It deals with the **look and feel** part of the web page. With it, developers can control the styles of fonts, colors of texts, spacing between paragraphs, sizes and layouts of columns, background colors and background images of websites, designs of overall layout, sizes of screens, display variations in multiple devices, and a lot of other amazing effects.

The **separation** of **HTML** from **CSS** makes it easier to **maintain sites**, share style sheets across pages, and tailor pages to different environments. This is referred to as the **separation** of **structure** (or content) from **presentation**.

CSS is a language standardised by the **W3C** (World Wide Web Consortium).

1.2.1 Including CSS in an HTML document

There are three different ways to use CSS in an HTML document: internal, external, and inline.

1.2.1.1 Internal CSS

Internal or embedded CSS requires adding <style> tag in the <head> section of the HTML document.

This CSS style is an effective method of styling a single page. However, using this style for multiple pages is time-consuming because it is necessary to put CSS rules on every page of the website.

Adding the code to the HTML document can increase the page's size and loading time.

Example

```
<!DOCTYPE html>
<html>
    <head>
        <title>CSS in the same HTML document</title>
        <style>
            p {color: red; font-size: 12pt;}
        </style>
    </head>
    <body>
        <p>Some text.</p>
    </body>
</html>
```

1.2.1.2 External CSS

With external CSS, the web pages link to an external .css file which can be created by any text editor.

This CSS type is a more efficient method, especially for styling a large website. By editing one .css file, you can change your entire site at once.

Since the CSS code is in a separate document, the HTML files will have a cleaner structure and are smaller. The same .css can be used for multiple pages.

The pages may not be rendered correctly until the external CSS is loaded.

Linking to multiple CSS files can increase the site's download time.

To link CSS to HTML, you must use the link tag with some relevant attributes. A **link** tag is a tag you should put at the **head** section of the HTML document.

The **link** tag has three important attributes: **rel**, **type** and **href**.

The **rel** attribute is used for the relationship between the external file and the current file. For CSS, you use **stylesheet**.

The **type** attribute is used for the type of the document you are linking to the HTML. For CSS, it is **text/css**.

href stands for "hypertext reference ". You use it to specify the **location** of the CSS file and the **file name**.

Example

```
<!DOCTYPE html>
<html>
    <head>
        <title>CSS in an external file</title>
        <link rel="stylesheet" type="text/css" href="css/styles.css"/>
    </head>
    <body>
        <p>Some text.</p>
    </body>
</html>
```

styles.css file content:

```
p {color: red; font-size: 12pt;}
```

1.2.1.3 Inline CSS

Inline CSS is used to style a specific HTML element. For this CSS style, you'll only need to add the style attribute to each HTML tag, without using selectors.

This CSS type is **not recommended**, as each HTML tag needs to be styled individually. Managing the website may become too hard using only inline CSS.

However, inline CSS in HTML can be useful in some situations. For example, in cases where you don't have access to CSS files or need to apply styles for a single element only.

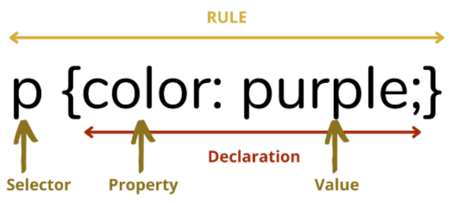
The **styles** are included in the **style** attribute of each **tag**.

Example

```
<!DOCTYPE>
<html>
    <head>
        <title>CSS inside HTML tags</title>
    </head>
    <body>
        <p style="color: red; font-size: 12pt; ">Some text.</p>
    </body>
</html>
```

1.2.2 Anatomy of a CSS rule

The different components of a CSS style are rule, selector, declaration, property, and value.



- **Rule**: Each of the styles that make up the stylesheet.
- **Selector**: Specifies the item or HTML elements on which the rule applies.
- **Declaration**: Specifies the styles that apply to the elements. Each declaration ends always on ;.
- **Property**: Feature modified in the selected item.
- **Value**: Sets the value of this feature.

Example

```
h1 {c
olor: red;
font-size: 16pt;}
```

1.2.3 Basic Selectors

One of the basic concepts of CSS is that of selectors. A selector determines which element(s) a CSS rule applies to.

A single CSS rule can apply to multiple elements.

The basic types of selectors are: universal, element, id, class and attribute.

1.2.3.1 The universal selector

The universal selector, specified as an asterisk (*), matches all elements. It can be specified as a single selector, to select all elements in the document, or with combinators.

Example

```
* {
        margin: 0;
  }
```

The CSS rule in the example will apply a margin of 0 to all elements in the document.

1.2.3.2 Element selectors

An element selector targets an HTML element by its tag name. The syntax of the selector is simply the name of the element.

Example

```
p {
   margin: 25px;
}
```

The CSS rule in the example will apply a margin of 25px to all p elements in the document.

1.2.3.3 ID selectors

An HTML element can have an id attribute. There should only be one element with a given id. An id selector is specified with the # character followed by the id value.

Example

```
#header {
        padding: 25px;
}
```

In the example, there is an element with an id attribute whose name is header that will receive 25px of padding.

1.2.3.4 Class selectors

An HTML element can also have a class attribute. A class can be used to mark all elements of a related type.

While only a single element is intended to be targeted by an id selector, any number of HTML elements can have the same class attribute. Similarly, a single HTML element can have any number of classes applied to it. Multiple classes are separated by a space in the value of the class attribute.

A class selector will match every element in the document with the given class. Class selectors are specified with a dot, followed by the name of the class.

Example

```
.nav-link {
color: darkcyan;}
```

In the example, the rule will match every element in the document with a class of nav-link and give it a colour of darkcyan.

1.2.3.5 Attribute selectors

HTML elements can also be selected by their attribute values or by the presence of an attribute. The attribute is specified inside square brackets, and the attribute selector can take several forms.

Examples

[name]

Selects all elements that have the given attribute, regardless of its value.

[name="value"]

Selects all elements that have the given attribute, whose value is the string value.

[name~="value"]

Selects all elements that have the given attribute, whose value contains the string value separated by white space.

[name*="value"]

Selects all elements that have the given attribute, whose value contains the substring value.

[name^="value"]

Selects all elements that have the given attribute, whose value begins with value.

[name$="value"]

Selects all elements that have the given attribute, whose value ends with value.

1.2.3.6 Compound selectors

Any of the preceding selectors (apart from the universal selector) can be used alone or in conjunction with other selectors to make the selector more specific.

Examples

div.my-class

Matches all div elements with a class of my-class.

span.class-one.class-two

Matches all span elements with a class of both class-one and class-two.

a.nav-link[href*="example.org"]

Matches all a elements with a class of nav-link that have an href attribute that contains the string example.org.

1.2.3.7 Multiple independent selectors

A CSS rule can have multiple selectors separated by a comma. The rule will be applied to any element that is matched by any one of the given selectors.

Example

.class-one, .class-two

Matches all elements with a class of class-one as well as all elements with a class of class-two.

1.2.3.8 Selector combinators

Combinators are used to select more specific element. Combinators are used in conjunction with the basic selectors. For a given rule, multiple basic selectors can be used, joined by a combinator.

Descendant combinator

The descendant combinator matches an element that is a descendant of the element on the left-hand side. Descendant means that the element exists somewhere within the child hierarchy – it does not have to be a direct child.

The descendant combinator is specified with a space character.

Example

.header div

Matches all div elements that are direct or indirect children of an element with a class of header. If any of these div elements have children that are also divs, those divs will also be matched by the selector.

Child combinator

The child combinator matches an element that is a direct child of the element on the left-hand side. It is specified with a > character.

Example

.header > div

Matches all div elements that are direct children of an element with a class of header.

General sibling combinator

The general sibling combinator matches an element that is a sibling, but not necessarily an immediate sibling, of the element on the left-hand side. It is specified with a ~ character.

Example

Consider the following HTML:

```html
<div>
    <div class="header"></div>
    <div class="body"></div>
    <div class="footer"></div>
</div>
```

The selector **.header ~div** would match two div elements: the one with class body and the one with class footer. It does not match the div with class header.

Adjacent sibling combinator

The adjacent sibling combinator is like the general sibling combinator, except it only matches elements that are an immediate sibling. It is specified with a + character.

The selector **.header + div** would only match the body element in the previous example, because it is the adjacent sibling of the heading element.

Using multiple combinators

Combinators can be combined to form even more specific selectors.

1.2.4 Pseudo-classes

A pseudo-class allows you to select elements based on some special state of the element.

Pseudo-classes start with a colon and can be used alone or in conjunction with other selectors.

Some pseudo-classes let you select elements based on UI state, while others let you select elements based on their position in the document.

1.2.4.1 UI state

These pseudo-classes are based on some UI state.

:active

Matches an element that is currently being activated. For buttons and links, this usually means the mouse button has been pressed but not yet released.

:checked

Matches a radio button, checkbox, or option inside a select element that is checked or selected.

:focus

Matches an element that currently has the focus. This is typically used for buttons, links and text fields.

:hover

Matches an element that the mouse cursor is currently hovering over. This is typically used for buttons and links but can be applied to any type of element.

:valid, **:invalid**

Used with form elements using HTML validation. The :valid pseudo-class matches an element which is currently valid according to the validation rules, and :invalid matches an element which is not currently valid.

:visited

Matches a link whose URL has already been visited by the user. To protect a user's privacy, the browser limits what styling can be done on an element matched by this pseudo-class.

1.2.4.2 Document structure

These pseudo-classes are based on an element's position in the document.

:first-child, :last-child

Matches an element that is the first or last child of its parent.

Example

Given the following HTML code:

```
<ul class="my-list">
    <li>Item one</li>
    <li>Item two</li>
</ul>
```

The selector .my-list > li:first-child will match the first list item only, and the selector .my-list > li:last-child will match the last item only.

:nth-child(n)

This pseudo-class takes an argument that matches an element that is the nth child of its parent. The index of the first child is 1.

Referring to the last example, we could also select the first item with the selector .my-list > li:nth-child(1), or the second item with the selector .my-list > li:nth-child(2).

The :nth-child pseudo-class can also select children at a given interval. For example, on a longer list, we could select every second child starting with the second child (2, 4, 6, 8 , 10, …) with the selector .my-list > li:nth-child(2n) or with .my-list > li:nth-child(even). We can select all odd-numbered children with the selector .my-list > li:nth-child(odd) or we could select every fourth child starting with the fourth child with the selector .my-list > li:nth-child(4n).

:nth-of-type(n)

Similar to :nth-child, except that it only considers children of the same type. For example, the selector div:nth-of-type(2) matches any div element that is the second div element that is the second div element among any group of children.

:root

Matches the root element of the document. This is usually the html element. This selector can be useful for several reasons, one of which is that it can be used to declare global variables.

1.2.4.3 Negating a selector

A selector can also include the **:not()** pseudo-class. :not accepts a selector as its argument and will match any element for which the selector does not match. For example, the selector div:not(.fancy) will match any div that does not have the fancy class.

1.2.4.4 Pseudo-elements

A pseudo-element lets you select only part of a matched element. Pseudo-elements are specified with a double colon followed by the pseudo-element name.

We haven't discussed block vs. inline elements yet, but it should be noted that some pseudo-elements only apply to block-level elements.

::first-line

Matches the first line of a block element.

::first-letter

Applies the styles only to the first letter of the first line of an element.

::before, ::after

Two special pseudo-elements are ::before and ::after. These pseudo-elements don't select part of the element. They actually create a new element as either the first child or the last child of the matched element, respectively. These pseudo-elements are typically used to decorate or add effects to an element.

Example

Suppose we want to add an indicator next to all external links on our website. We can tag these external links using a class, say, external-link.

We can specify an external link as shown:

```
<a class="external-link"
href="https://www.iessanclemente.net/">
IES San Clemente</a>
```

Then we can add the indicator with the CSS rule. The content property defines what the text content of the pseudo-element should be.

```
.external-link::after {
    content: ' (external)';
    color: green;
}
```

The ::after pseudo-element added the content (external) and made it green.

1.2.5 Specificity

CSS means Cascading Style Sheets. Cascading means that the **order** in which **CSS rules** are applied to an element **matters**.

An HTML element can have multiple CSS rules applied to it by matching different selectors. What happens if two or more of the rules applied to an element contain the same CSS property? How are such conflicts resolved?

Suppose the following HTML:

```html
<div class=profile>My Profile</div>
```

Suppose it is given the following styles:

```css
.profile {
   background-color: green;
}
div {
   background-color: red;
   color: white;
}
```

We have a **conflict**. Our HTML element matches both selectors. Each rule specifies a different value for the background-color property. When the page is rendered, which background colour will the div element have?

When there is a conflict of CSS properties across multiple rules, the rule with the most specific selector will be chosen. According to the rules of CSS, a class selector is more specific than an element selector. Specificity rules only matter for conflicting properties across multiple rules. Other properties in these multiple rules will still be applied.

The **specificity** rankings of CSS rules are as follows, from most specific to least specific:

1) Inline styles in an element's style attribute

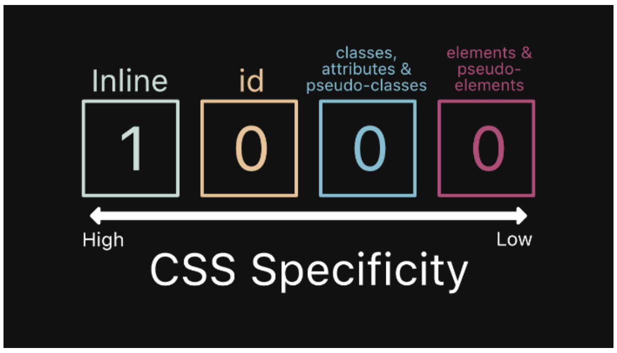
2) Id selectors

3) Class selectors, attribute selectors, and pseudo-classes

4) Element selectors and pseudo-elements

Neither the universal selector nor combinators factor into specificity.

Here is a visual representation of CSS specificity hierarchy:


Inline
id
classes, attributes & pseudo-classes
elements & pseudo-elements
1
0
0
0
High
Low
CSS Specificity

There is a general **algorithm** for calculating a CSS rules's specificity. To calculate the specificity of a CSS rule, imagine four boxes, one for each type of style rule in the give list. Initially, each box has a zero in it.

If the element has an inline style, add a 1 to the first box. In this case, the inline style automatically wins.

For each id in the selector, add 1 to the value in the second box. For each class, pseudo-class, or attribute in the selector, add 1 to the value in the third box. Finally, for each element or pseudo-element in the selector, add 1 to the value in the last box.

Lastly, if multiple conflicting rules are calculated to have the same specificity, the rule that appears last will win.

Any CSS property can have the keyword **!important** after it inside of a rule. This keyword will cause that property to always win in a conflict.

However, this is generally considered a poor practice. It can make CSS issues harder to debug and can make the style sheets less maintainable.

1.2.6 CSS box model

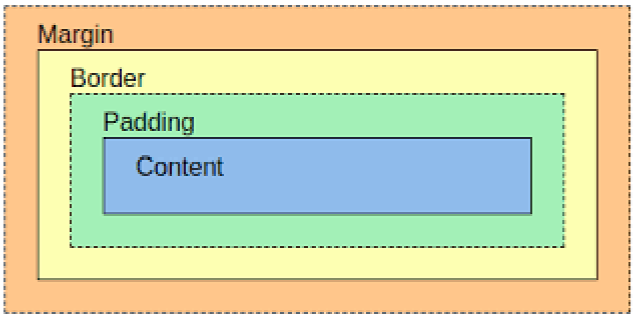
Every element in CSS is treated like a rectangular box. This is sometimes referred to as the box model. The box is made up of four parts. Starting from the outside and moving toward the centre, these are the margin, border, padding and content.

The **margin** is the space between an element's border and its surrounding elements. It is specified with the margin property.

The **border** is an outline around the box. Borders can be styled with a thickness style and colour. It is specified with several properties: border-style, border-width, border-color, and border.

The **padding** is the space between the element's border and the content itself. It is specified with the padding property.

By default, most elements have no padding, border, or margin. There are some exceptions, like button elements.



1.2.6.1 Box sizing

The **size** of an **element** is specified with the **width** and **height** properties. How exactly this is interpreted, however, depends on the value of the **box-sizing** property. This property supports two values, **content-box** and **border-box**.

content-box

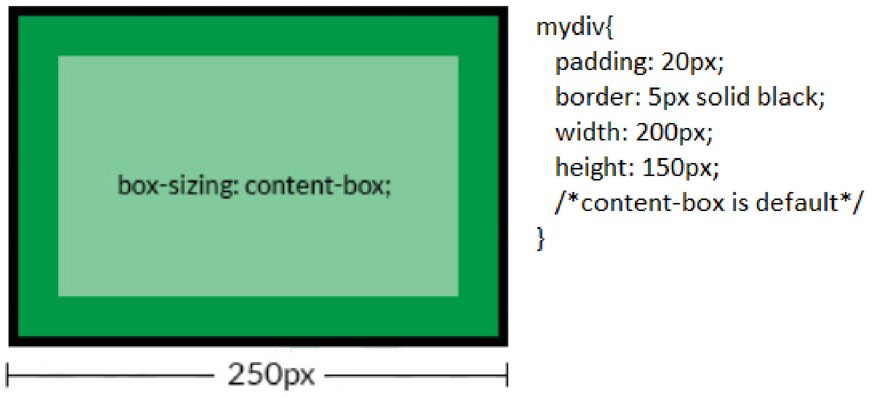
This is the default. With content-box, the width and height properties are treated as the width and height of the content area of the box only. The actual width and height taken up by the element's box is the sum of the specified width and height (the content box), the padding on each side, and the border width on each side.

Example

You have a 200px by 150px div and you want to add 20px of padding and a 5px border to it.

```
mydiv {
    padding: 20px;
    border: 5px solid black;
    width: 200px;
    height: 150px;
}
```

You get a 250px by 200px container.



```
mydiv{
  padding: 20px;
  border: 5px solid black;
  width: 200px;
  height: 150px;
  /*content-box is default*/
}
```

border-box

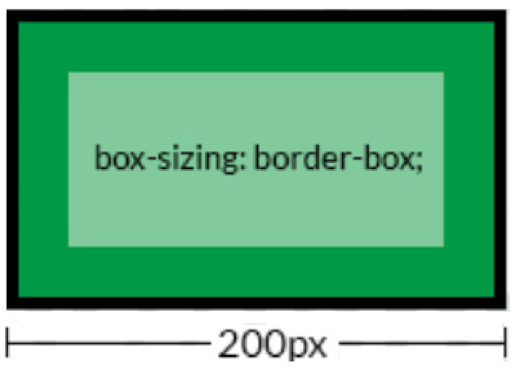
With border-box, the values of the width and height properties are treated as the size of the content box plus the padding and border width.

Example

The following CSS:

```
mydiv {
    padding: 20px;
    border: 5px solid black;
    width: 200px;
    height: 120px;
    box-sizing: border-box;
}
```

Looking back at the previous example, and setting box-sizing to border-box, the total width and height of the rendered element is 200 pixels. To compensate for the extra 50 pixels taken up by the padding and border width, the size of the content box will shrink to 150 pixels.



```
mydiv{
  padding: 20px;
  border: 5px solid black;
  width: 200px;
  height: 150px;
  box-sizing: border-box;
}
```

1.2.7 Block and inline elements

There are two types of HTML elements: **block** and **inline** (or a combination of the two, **inline-block**). Both block and inline elements follow the box model but are different in some important ways.

Some HTML elements are block element (e.g., div) and some are inline elements (e.g., span). An element's type can be changed by setting the **display** property to **block**, **inline** or **inline-block**.

1.2.7.1 Block elements

A block element always appears on its own line and takes up the full width of its containing element, unless an explicit width is set with the width property. The height of a block element, by default, is just enough to fit the height of its content, but this height can also explicitly be set with the height property.

Example of block elements are: <div>, <p>, <li>, <main>, <nav>, <ul>, <form>, <table>, <aside>, <article>.

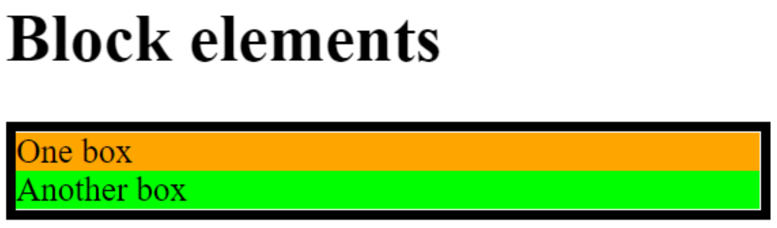
Example

The following HTML:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>Block elements</title>
        <style>
            .container {
                width: 350px;
                border: 5px solid black;
            }
            .box1 {
                background-color: orange;
            }
            .box2 {
                background-color: lime;
            }
        </style>
    </head>
    <body>
        <h1>Block elements</h1>
        <div class="container">
           <div class="box1">One box</div>
           <div class="box2">Another box</div>
        </div>
    </body>
</html>
```

The outer container element has an explicit width of 350px set, so it will be 350 pixels wide (technically, 360pixels, since it is using content-box sizing).

The inner elements have no explicit width set, so they take up the full width of the container element. They also have no explicit height set, so they only take up enough vertical space to fit the text content.


Block elements
One box
Another box

1.2.7.2 Inline elements

Unlike block elements, an inline element is rendered inside the normal flow of text. They only take up enough width and height as necessary to contain their content. Setting the width or height properties of an inline element will have no effect.

Example of inline elements are: <a>, <strong>, <em>, <span>.

Example

The span is an inline element, so its width and height are only enough to fit its content. It does not appear on its own line.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>Inline elements</title>
        <style>
            .highlight {
                background-color:dodgerblue;
            }
        </style>
    </head>
    <body>
        <p>The element <span class="highlight">span</span> is an inline element.</p>
    </body>
</html>
```

The element span is an inline element.

1.2.7.3 Inline-block elements

The third element type is a combination of the first two. An inline-block element flows with the text like an inline element, but the width and height properties are respected, as are the vertical padding and margin.

Inline block elements cannot contain block-level elements.

1.2.8 Units of measurement

CSS allows you to specify **absolute** or **relative** units of measurement.

Some of the **absolute** units are: in (inches), cm (centimetres), mm (millimetres), pt (points) and px (pixels).

In CSS, 1 **pixel** is formally defined as **1/96** of an **inch**. All other absolute length units are based on this definition of a pixel.

Example

```
<!DOCTYPE html>
<html>
    <head>
        <style>
            p {
               font-size: 44px;
            }
        </style>
    </head>
    <body>
        <p>This is a paragraph</p>
    </body>
</html>
```

A **relative** unit gets sizing from something else. In the specification, the relative length units are defined as **em**, **ex** and **rem**. These are font-relative lengths. The specification also defines a **%** value, which is always relative to another value. Using relative values means that things can scale up and down according to some other value.

The CSS **em** unit gets its name from a typographical unit. In typography, the term em was originally a reference to the width of the capital M in the typeface and size being used. The em unit is relative to the current font size of the element.

When used with the font-size property, em inherits the font-size from its parent element.

The em unit is practical for creating entirely scalable and flexible layouts.

Example

```
<!DOCTYPE html>
<html>
    <head>
        <style>
            body {
               font-size: 20px;
               background: purple;
            }
            .big {
               font-size: 2em;
               background: red;
            }
            .small {
               font-size: 0.5em;
               background: darksalmon;
            }
        </style>
    </head>
    <body>
        The Exact font size of the body is 20px.
        <p class="big">The font size of the paragraph is 2*20px=40px <p>
        <p class="small">The font size of the paragraph is 0.5*20 = 10px</p>
    </body>
</html>
```

The **ex** unit is relative to the x-height of the font. It is rarely used.

The **rem** unit is relative to the font size of the root element. As with em, the rem unit will scale but with the root element instead of the parent.

Example

```
<!DOCTYPE html>
<html>
    <head>
        <style>
            html {
               font-size: 40px;
            }

            .text {
               font-size: 2rem;
            }
        </style>
    </head>
    <body>
        <p class="text">This is a paragraph</p>
        <p>This is another paragraph</p>
    </body>
</html>
```

rem units are good choice, especially for layout properties, since the size of 1rem remains constant throughout the document. If the browser is zoomed, everything resizes nicely because it's all proportional to the base font size.

Percentage (%) units are relative units based on a 100% scale.

When used for things like font size, percentages are relative to the containing element type size. So setting a font size of 200% of default body type (the default text size in browsers is 16px) would result in a font size equal to 32px.

When used for length measurements like width or margin, percentages are relative to the containing element.

Percentage lengths are great for sizes that scale relative to a user's browser size. They are very useful in responsive design.

Example

```
<!DOCTYPE html>
<html>
    <head>
        <style>
            p {
                width: 50%;background-color:yellow;
            }
            strong{ font-size: 150%;}
        </style>
    </head>
    <body>
        <p>There are <strong>important</strong> challenges ahead of us.</p>
    </body>
</html>
```

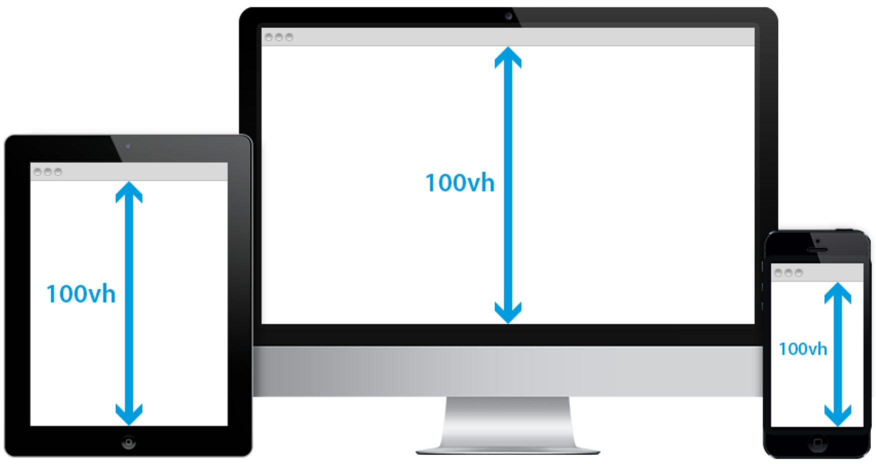
Viewport units: vw and vh

The viewport is the area of the page that is currently visible in your web browser. View port units are percentages of the size of the browser window. There are two different viewport unit value types:

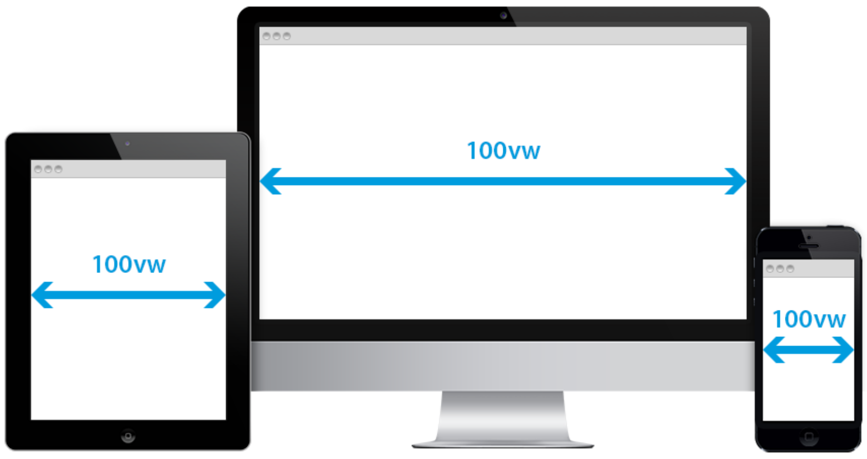
- **vw** (View Width): which is relative to the width of the browser window.
- **vh** (View Height): which is relative to the height of the browser window.

If the viewport is resized, the any elements using vw units will have their sizes adjusted accordingly. Because vw and vh are relative to the viewport size, they are a good choice when using responsive design techniques.

There are also two related units, **vmin** and **vmax**. Vmin is defined as whichever is smaller – the viewport width or the viewport height – and vmax is the larger of the two.



1.Viewport height



2. Viewport width

Some property values take **no units** at all but rather just a number.

1.2.9 Colours

One of the most common things done with CSS is to change colours. This can include background colour, text colour and border colour. There are a multitude of colours, and they can be expressed in multiple ways.

1.2.9.1 Predefined colours

CSS has many predefined colour values (145). A full list of these colour can be found at: https://wtool.org/css-named-colors/

1.2.9.2 RGB colours

One of the ways to define a colour is by the values of the colour's red, green, and blue components. Any colour can be expressed as a combination of RGB values. Each value of red, green, and blue is expressed as a number between 0 and 255.

One way that an **RGB colour** can be specified is as a **hexadecimal** value. The red, green, and blue values are each converted to two hexadecimal digits. These digits are used in RGB order, preceded by a pound sign (#). The hexadecimal values can be specified with uppercase or lowercase letters.

The other way to specify an RGB colour is by using the **rgb function**. Instead of hexadecimal digits, the red, green, and blue components of the colour are specified as base-10 numbers between 0 and 255 or a percentage between 0% and 100%.

Thus, to specify white using percentage notation, the values would be: rgb(100%,100%, 100%) and using the integer-triplet notation would be: rgb(255,255,255).

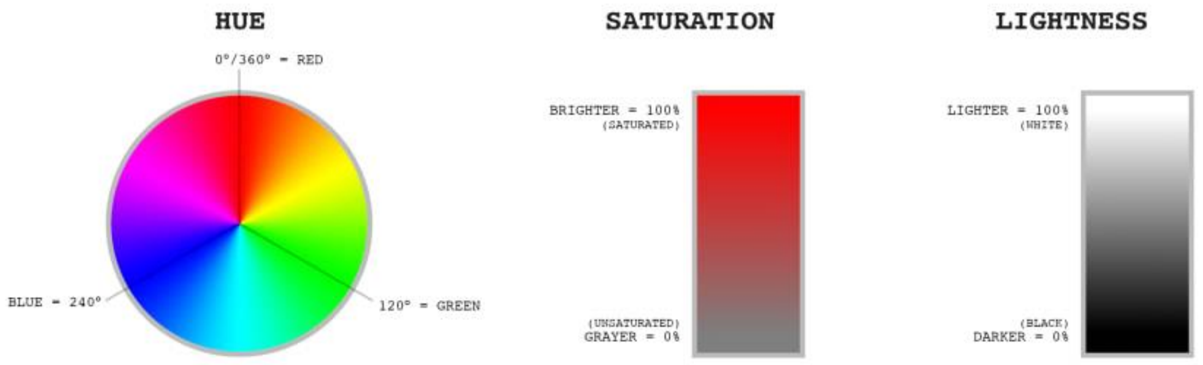
RGB colours can also specify an **alpha value**, which determines the **opacity** of the colour. The alpha is a value between 0 (fully transparent) and 1 (fully opaque) or a percentage between 0% and 100%. To specify an alpha value, the **rgba function** is used. For example, for pure red with 50% opacity, the colour would be defined as rgba(255, 0, 0, 0.5).

1.2.9.3 HSL colours

A colour can also be expressed as a combination of hue, saturation, and lightness values. **Hue** is specified as a degree of an angle on the colour wheel (from 0 to 360 degrees). = degrees is red, 120 degrees is green, and 240 degrees is blue.

Saturation is a percentage value of how much colour is applied. 0% saturation is a shade of gray, and 100% saturation is the full colour from the colour wheel.

Finally, **lightness** is also a percentage value. 0% lightness is pure black, and 100% is pure white.



An HSL colour is specified using the **hsl function**. For the colour with a hue of 120 degrees, a saturation of 50% and a lightness of 50% would be specified as hsl(120, 50%, 50%).

Like RGB colours, HSL colours can also have an alpha value, specified using the **hsla function**. For the color with a hue of 120 degrees, a saturation of 50%, a lightness of 50% and an opacity of 75% it would be specified as hsla(120, 50%, 50%, 0.75).

1.2.9.4 Newer colour syntax

The usage of the rgb/rgba and hsl/hsla functions as previously described has been the standard for a long time. There is a newer syntax for these functions that is slightly different.

This newer format is completely optional, and in fact, if you have to support older browsers, you won't want to use it.

The new syntax makes a few changes:

- There are no commas between the numbers.
- If an alpha value is specified, it is separated from the three colour values with a slash.
- The alpha value can also be specified on a hex colour.

For example, #FF00007F represents pure red with 50% alfa and rgb(255 0 0 /0.5) represents the same colour.

1.2.9.5 Transparent

Anywhere a colour is expected, the transparent keyword can be used. This will apply no colour.

1.2.10 Comments

Comments in CSS begin with /* and end with */

```
/* This is a CSS comment */
/* This is a
Multiple line
CSS comment */
```

1.2.11 Text styling

One of the most common uses for CSS is to **style text**. Most webpages include text, after all, and changing the look of it can go a long way toward giving a webpage a more unique appearance. Without changing the HTML underneath, CSS can be used to alter the **size of text**, the **font**, the **boldness**, the **alignment** within a paragraph, and more.

Fonts

CSS has several properties beginning with font-, which allow you to control many features of the text characters (or glyphs) themselves.

For example, to apply Arial font we can use the declaration: font-family: Arial;

font-family allows you to specify which font, or fonts, a selection of elements will use.

There are about 11 fonts that are installed across almost all systems, termed **web safe fonts** because they are safe for use in your pages. The list includes:

- **Sans-serif**. Font without serifs: Verdana, Arial, Trebuchet MS.
- **Serif**. Fonts with serifs: Times new roman, Georgia.
- **Monospaced**. Fonts in which every glyph takes up the same space: Courier new
- **Cursive**. Fonts that have a decorative, often handwritten looking style: Comic Sans

- **Fantasy**. Fonts that have a bold, often ornamental, or quirky style, which are meant to be used for headings: Impact.

If you want you can apply several fonts to a single element selection, known as **font stack**.
Example: font-family: 'Helvetica neue', arial, verdana, sans-serif;
When you specify a font stack like this, the browser goes through them from left to right, until it finds a font that is installed on the system, and therefore can be used.
Note that fonts with more than one word in their names must be surrounded by quotes.
It is often a good idea to use a different font than the one used in the body for at least some of the headings, to make them stand out and give the site a bit more character.
Web fonts are other interesting feature that allow us to specify our own custom font files to download with our web pages. This completely gets around the problem of fonts not being available on users' machines. To specify a web font for download on a page, you reference the font in a special @font-face block that goes at the top of the style sheet and looks something like this:

```
@font-face {
  font-family: 'My font';
  src: url('myfont.ttf') format('truetype');
}
```

The following URL has fonts to download: https://www.dafont.com/, https://fonts.google.com/.
The following URL is a @font-face generator: https://transfonter.org/
The property **font-size** allows you to set the size of the text inside selected elements, using any CSS unit available, such as pixel, em, percentage and more.
You can also use size keywords xx-small, x-small, small, medium, large, x-large and xx-large. These tend to be best used when you want to set font sizes relative to each other.
Example: font-size: 62.5%;
The property **font-weight** can be used when you want text to be bolder. Possible values are:

- **bold** and **bolder** provide two levels of extra boldness.
- **lighter** provides a level of less boldness.
- **100**, **200**, … all the way up to **900** provide incremental levels of boldness.
- **normal** is the default non-bold setting.

Most of these settings don't really make a visual difference, especially when setting small font sizes, and when the font you are using doesn't have different levels of boldness defined in it.

The **font-style** property can take the values italic, oblique, and normal. Normal is the default, italic tells the browser to use the italic version of the font (if available), and oblique tells the browser to use the normal version of the font.
Example: font-style: italic;
In CSS, the **background-color** property is pretty straightforward for setting the background color of anything.
The property in CSS to set the foreground color is **color**.
CSS **text align** property is used to align text in left, center, right and justify. Default text align is left. Only block level elements support text-align property. That means p, headings h1-h6, blockquote, elements support text align, but inline level elements doesn't. But inline can inherit text align from parent element.
Example: text-align:center;
Text indent property is used to change default indentation of css text. **text-indent** changes the position of first word of first line in right direction or left direction. If text indent is negative, it will move towards left and overflow from document. Try to avoid text indent in negative.
Example: text-indent:-50px;
CSS **text decoration** add or remove decoration of text like **underline**, **overline**, **line-through** and **none**. Text decoration can have text decoration **line**, text decoration **colour** and text decoration **style**.
CSS **text transform** property is used to transform text from lowercase to uppercase, lowercase to capitalise and uppercase to lowercase for alphabets.
Example: text-transform:uppercase;
Word break property of CSS is used to specify whether to break lines within words or not. Default value is **normal**. Another value is **break-all**.
Example: word-break: break-all;
Word spacing is spacing between words. Default value of word spacing is 0px. Word spacing can have both positive and negative values.
Example: word-spacing:0px;
Letter spacing is space between two letters. Default value of letter spacing is 0px. Letter spacing can have both positive and negative values.
Example: letter-spacing:10px;

The **line-height** property defines the amount of space above and below inline elements. The line-height property can accept the keyword values normal or none as well as a number, length, or percentage.

The default value is normal. Usually this means that it is set to 1.2, this depends on the browser vendor. A number value without any unit means that it is only a multiplier. It takes the font-size value and multiplies it by 1.2.

Example: line-height:1.35;

1.2.12 CSS positioning

When you want to design complex layouts, you'll need to change the typical document flow and override the default browser styles.

You must control how elements behave and are positioned on the page.

The CSS property to control elements positioning is **position**.

This property takes in five values: **static**, **relative**, **absolute**, **fixed** and **sticky**.

1.2.12.1 Position static

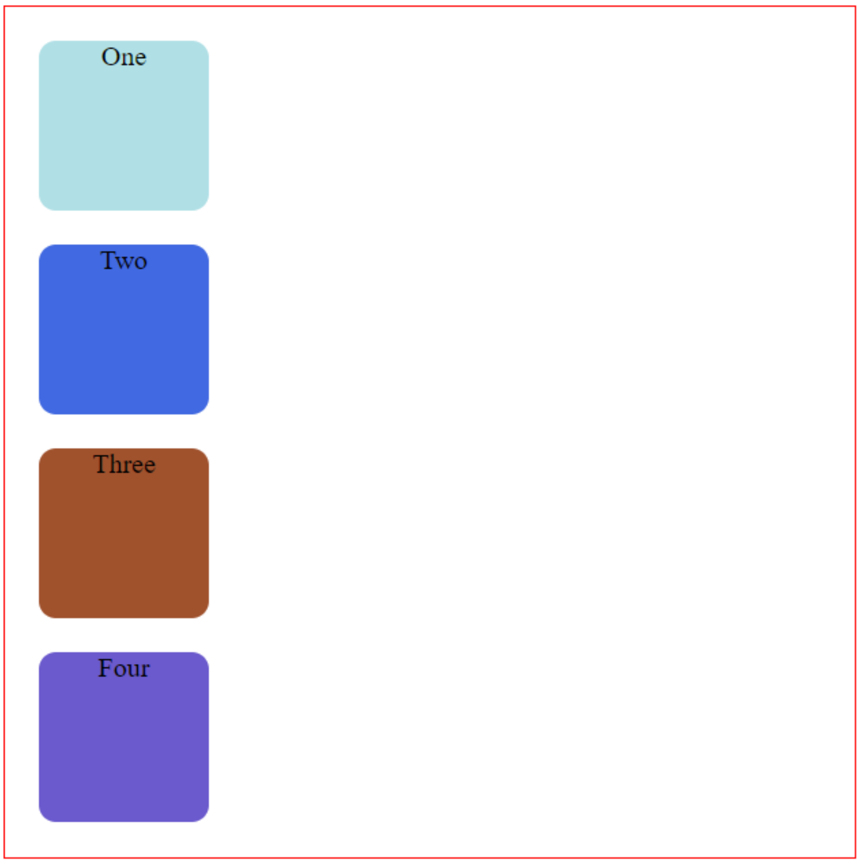
By **default**, the position property for all HTML elements in CSS is set to **static**. Visually, all elements follow the order of the HTML code, and in that way, the typical document flow is created. Block elements are stacked one after the other.

Example

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <link rel="stylesheet" href="style.css">
        <title>CSS Positioning</title>
        <style>
            body {
            margin: 100px auto;
            }

            .parent {
            width: 500px;
            border: 1px solid red;
            margin: auto;
            text-align: center;
            }

            .child {
            border-radius: 10%;
            width: 100px;
            height: 100px;
            margin: 20px;
            }

            .one {
```

```
            background-color: powderblue;
            }

            .two {
            background-color: royalblue;
            }

            .three {
            background-color: sienna;
            }

            .four {
            background-color: slateblue;
            }

        </style>
    </head>
    <body>
        <div class="parent">
            <div class="child one">One</div>
            <div class="child two">Two</div>
            <div class="child three">Three</div>
            <div class="child four">Four</div>
        </div>
    </body>
</html>
```

The browser shows the following:



Whatever comes first in the HTML is shown first, and each element follows the next, creating the document flow.

This default positioning doesn't leave any room for flexibility or to move elements around.

1.2.12.2 Position relative

The position **relative** works the same way as position static, but it lets you change an element's position. To modify the position, you'll need to apply the top, bottom, right and left properties.

The top, bottom, right and left offsets push the tag away from where it is specified, working in reverse.

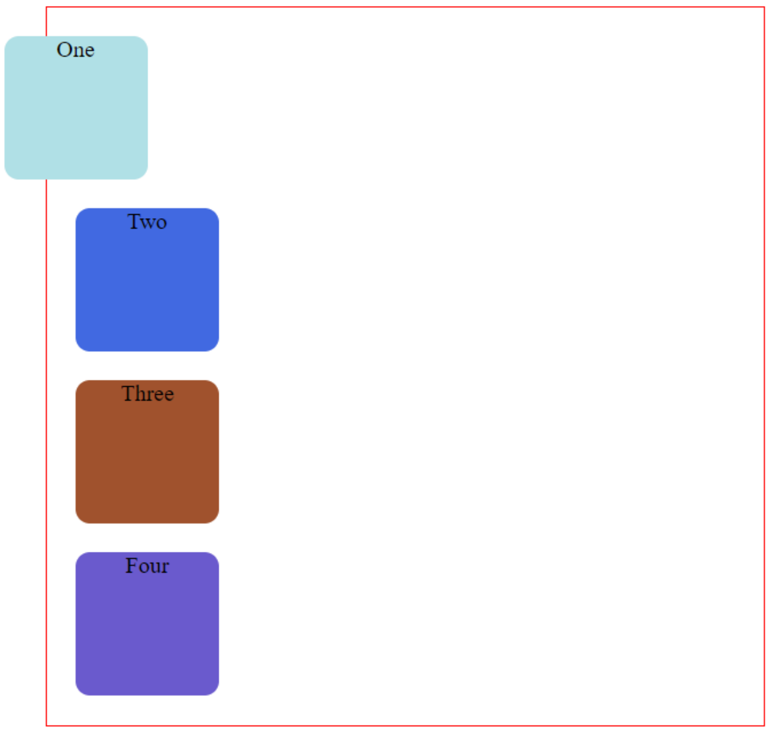
top in fact moves the element towards the bottom of the element's parent container. **bottom** pushes the element towards the top of the element's parent container, and so on.

Example

To move the first square in last example to the left, you can update the CSS like this:

```
.one {
  background-color: powderblue;
  position: relative;
  right: 50px;
}
```

Here, the square moves 50px from the left of where it was supposed to be by default.



The **position:relative**; changes the position of the element relative to the parent element and relative to itself and where it would usually be in the regular document flow of the page. This means that it's relative to its original position within the parent element.

1.2.12.3 Position absolute

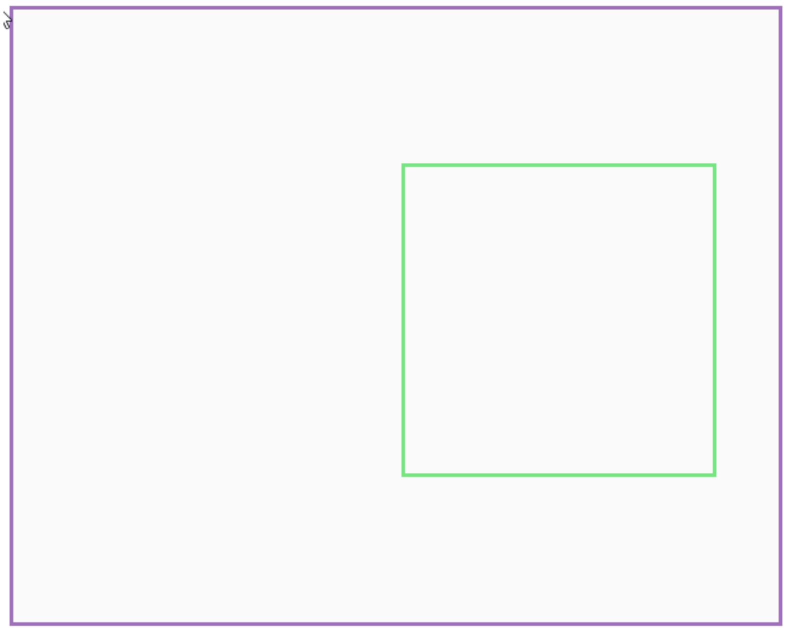
Position **absolute** means position of an HTML element in a specific location outside of normal document flow. An element with position absolute will be positioned with respect to its nearest non-static ancestor. In the case when there is no positioned parent element, it will be positioned related directly to the HTML element (the page itself). The positioning of an element does not depend upon its siblings or the elements which are at the same level.

An element with absolute position can be placed anywhere on the page using properties top, left, right and bottom. Absolut element can also overlap other elements by using **z-index** property.

Example

```html
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <title>Position absolute</title>
        <style>
            #parent {
            /*position: relative; */
            width: 500px;
            height: 400px;
            background-color: #fafafa;
            border: solid 3px #9e70ba;
            font-size: 24px;
            text-align: center;
            }

            #child {
            position: absolute;
            right: 40px;
            top: 100px;
            width: 200px;
            height: 200px;
            background-color: #fafafa;
            border: solid 3px #78e382;
            font-size: 24px;
            text-align: center;
            }
        </style>
    </head>
    <body>
        <div id="parent">
            <div id="child"></div>
        </div>
    </body>
</html>
```

The browser shows the following:



1.2.12.4 Position fixed

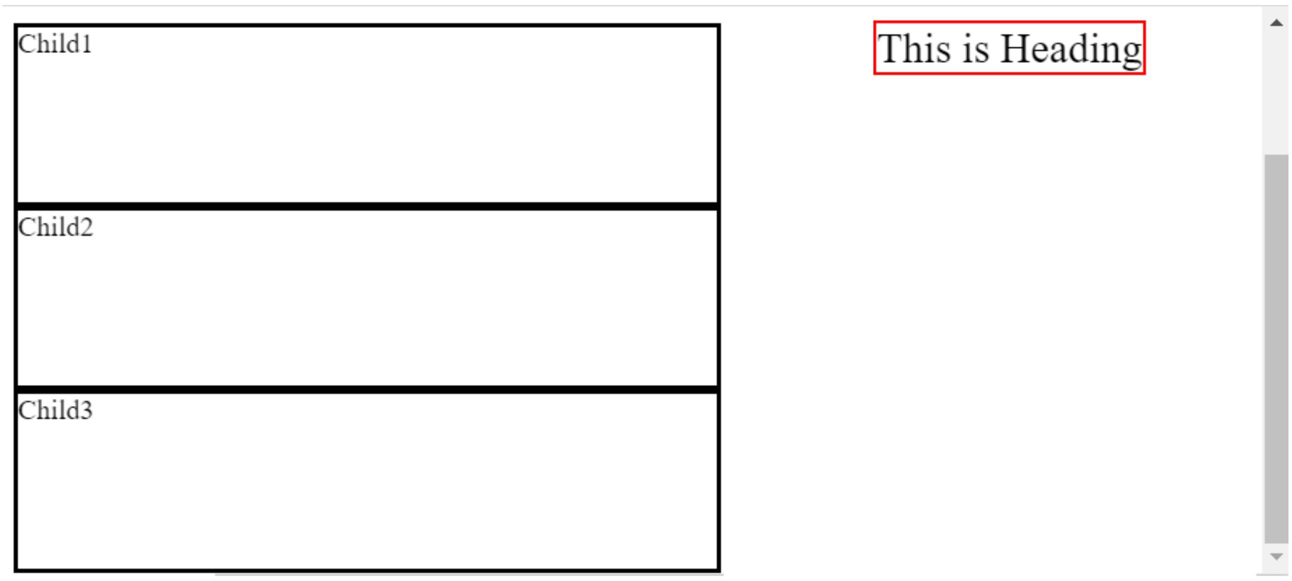
Elements with fixed positioning stay in the same position in the browser window even when the page is scrolled. It can be used to keep an element (for example a navigation menu) on the page at all times. Fixed position is often confused with absolute position, but while their behaviour is somewhat similar, they are not the same thing. An element with a fixed position is out of the flow of the rest of the document.

Example

```
<!DOCTYPE html>
<html>
    <head>
        <title>CSS Fixed Position</title>
        <style>
            .divChild {
            width: 400px;
            height: 100px;
            border: solid;
            position:relative;
            }

            #divHeading {
            border: 2px solid red;
            font-size:x-large;
            position:fixed;
            left:500px;
            }

            #divParent {
            position:relative;
            top:100px;
            }
        </style>
    </head>
    <body>
        <div id="divHeading">This is Heading</div>
        <div id="divParent">
```

```
            <div class="divChild">Child1</div>
            <div class="divChild">Child2</div>
            <div class="divChild">Child3</div>
        </div>
    </body>
</html>
```

The browser shows the following after scrolling:

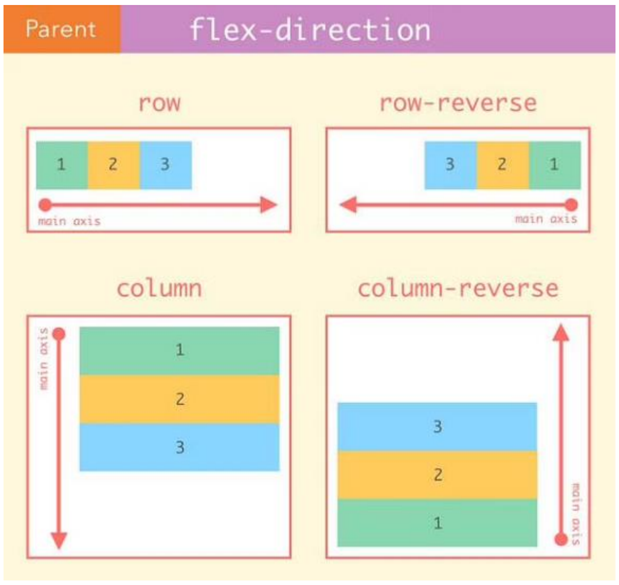


1.2.12.5 Flexbox

The Flexible Box Layout Module, more commonly known as flexbox, is a powerful tool for building layouts with CSS. Flexbox is a one-dimensional layout that can lay out elements either horizontally or vertically (but not both). An element using flexbox as its layout is referred to as a **flex container**, and the elements inside it are **flex items**.

A flex container has a **direction**, defined by the **flex-direction** property. It can be either a **row** (horizontal) or **column** (vertical). There are four values for the flex-direction property: row, row-reverse, column, and column-reverse.

A flex container is created by setting an element's **display** property to the value **flex**. This will create a block flex container. You can also create an inline flex container by setting display to **inline-flex**.



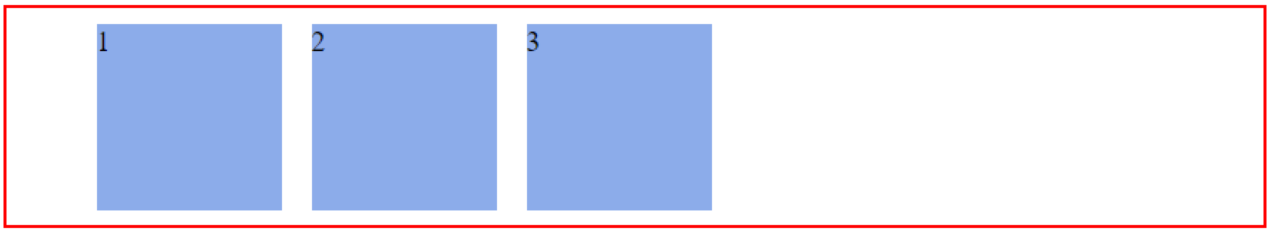
Related to the direction is the concept of the axis. The **main axis** points along the direction specified in flex-direction, and the **cross-axis** points perpendicular to it. For example, if flex direction is set to row, the main axis goes horizontally from left to right, and the cross axis goes vertically from top to bottom.

By default, the flex-direction property is set to row and it aligns the flex-items along the main axis.

Example

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <link rel="stylesheet" href="style.css">
        <title>CSS flexbox</title>
        <style>
            /*Make parent element a flex container*/
            ul {
            display: flex; /*or inline-flex*/
            border:2px solid red;}
            li {
            width: 100px;
            height: 100px;
            background-color: #8cacea;
            margin: 8px;
            list-style-type:none;
            }
        </style>
    </head>
    <body>
    <ul> <!--parent element-->
        <li>1</li> <!--first child element-->
        <li>2</li> <!--second child element-->
        <li>3</li> <!--third child element-->
    </ul>
    </body>
</html>
```

Here is what you should have:



The list elements are now stacked horizontally, from left to right.

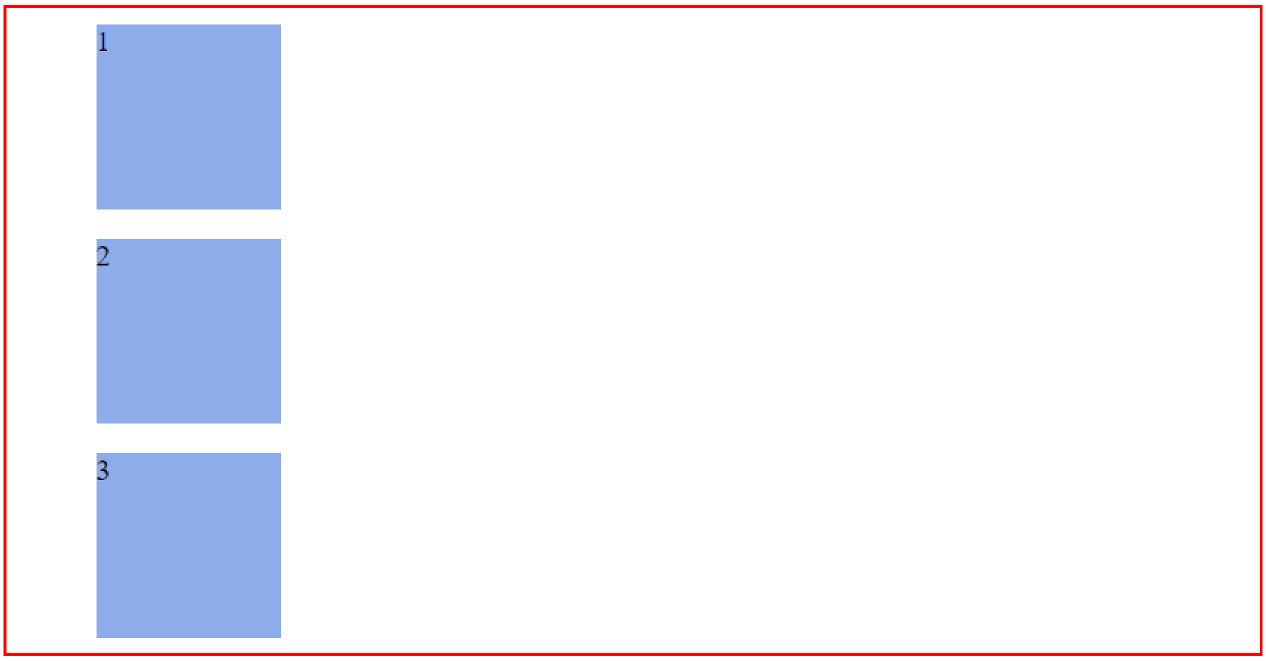
As soon as you set the display property to flex, the unordered list automatically becomes the flex container and the child elements (in this case, the list elements li) become flex items.

If the **flex-direction** property is changed to **column**, the flex-items will be aligned along the cross axis. They would stack from top to bottom, not from left to right any longer.

Example

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <link rel="stylesheet" href="style.css">
        <title>CSS flexbox column</title>
        <style>
            /*Make parent element a flex container*/
            ul {
            display: flex; /*or inline-flex*/
            flex-direction:column;
            border:2px solid red;
            }
            li {
            width: 100px;
            height: 100px;
            background-color: #8cacea;
            margin: 8px;
            list-style-type:none;
            }
        </style>
    </head>
    <body>
    <ul> <!--parent element-->
        <li>1</li> <!--first child element-->
        <li>2</li> <!--second child element-->
        <li>3</li> <!--third child element-->
    </ul>
    </body>
</html>
```

The following image shows the change:



The **flex-wrap** property controls whether the flex container is a **single-line** or **multi-line**, and the direction of the cross-axis which determines the direction in which new lines are stacked.

The flex-wrap property **default** value is **nowrap**. The flex container is a single-line.

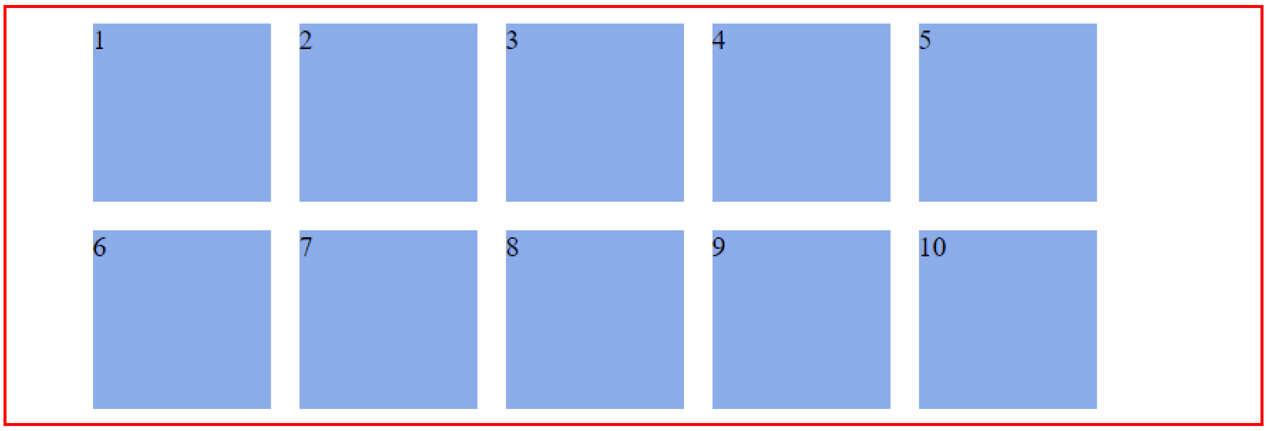
There are several values for the flex-wrap property: **nowrap**, **wrap**, **wrap-reverse** and **initial**.

The property **wrap** is used to break the flex items into multiple lines. It makes flex items wrap to multiple lines according to flex item width.

Example

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <link rel="stylesheet" href="style.css">
        <title>CSS flexbox wrap</title>
        <style>
            /*Make parent element a flex container*/
            ul {
            display: flex; /*or inline-flex*/
            flex-wrap:wrap;
            border:2px solid red;
            }
            li {
            width: 100px;
            height: 100px;
            background-color: #8cacea;
            margin: 8px;
            list-style-type:none;
            }
        </style>
    </head>
    <body>
    <ul> <!--parent element-->
        <li>1</li> <!--first child element-->
        <li>2</li> <!--second child element-->
        <li>3</li> <!--third child element-->
        <li>4</li>
        <li>5</li>
        <li>6</li>
        <li>7</li>
        <li>8</li>
        <li>9</li>
        <li>10</li>
    </ul>
    </body>
</html>
```

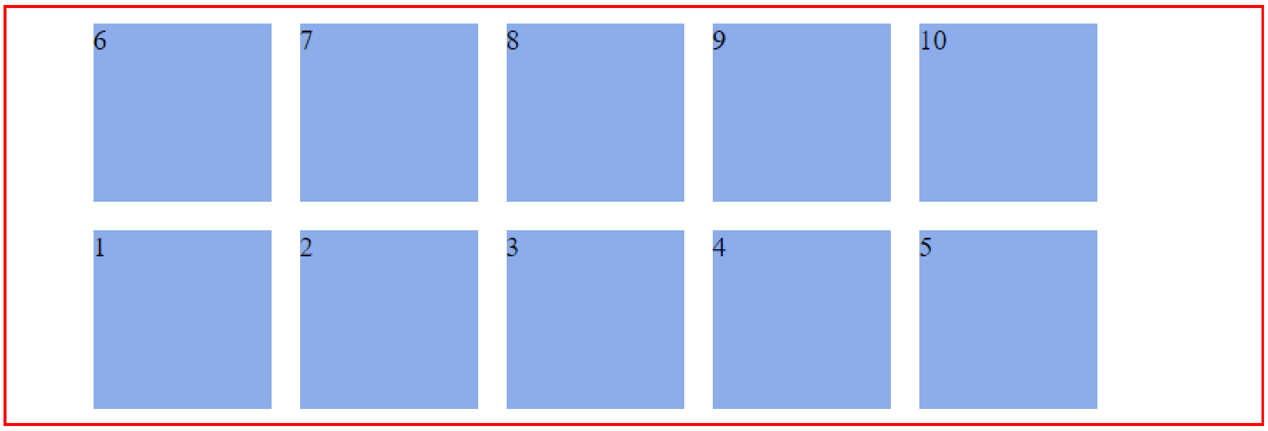
The browser shows two lines if necessary:



The property **wrap-reverse** is used to reverse the flow of the flex items when they wrap to new lines.

Example

We change the flex-wrap to wrap-reverse in last example and the browser shows:



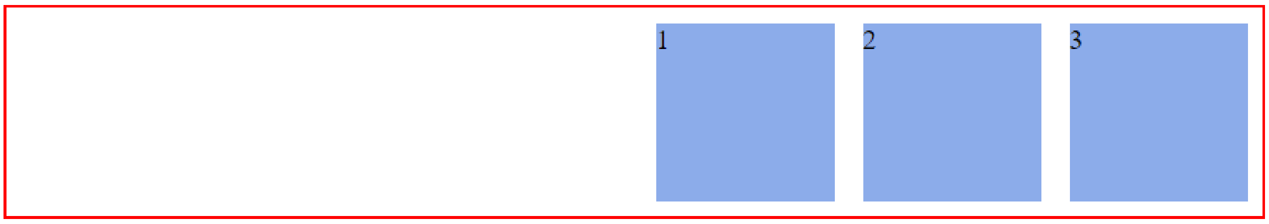
The **flex-flow** is a **shorthand property** which takes flex-direction and flex-wrap values.

Example: flex-flow: row wrap;

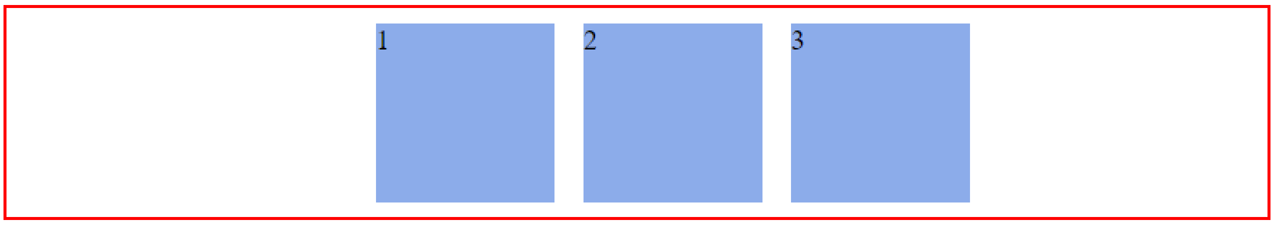
The **justify-content** property defines how flex items are laid out on the main axis. The justify-content property takes on any of the following 5 values: **flex-start**, **flex-end**, **center**, **space-between** and **space-around**.

The default value is **flex-start**. The property flex-start groups all flex-items to the start of the main axis.

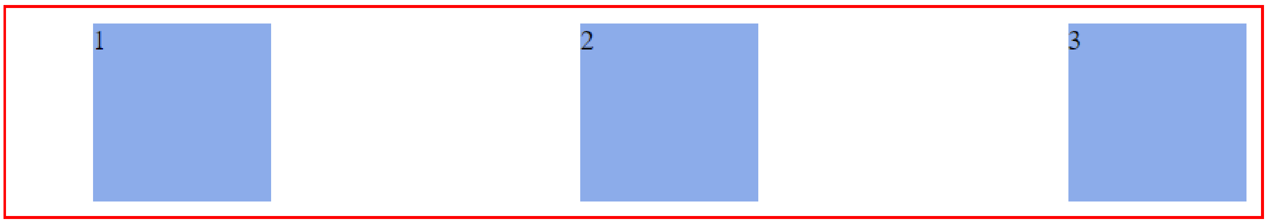
The value **flex-end** groups flex-items to the end of the main axis.



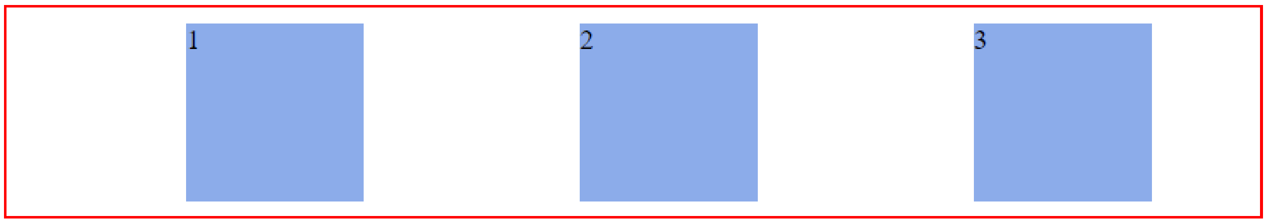
The value **center** centres the flex items along the main axis.



The value **space-between** keeps the same space between each flex item.



Finally, **space-around** keeps the same spacing around flex items.



The **align-items** property is set on the container. It controls how flex items are aligned along the cross axis. The default value is stretch, which will stretch items along the **cross axis** to fill the container's size, while respecting height and width constraints.

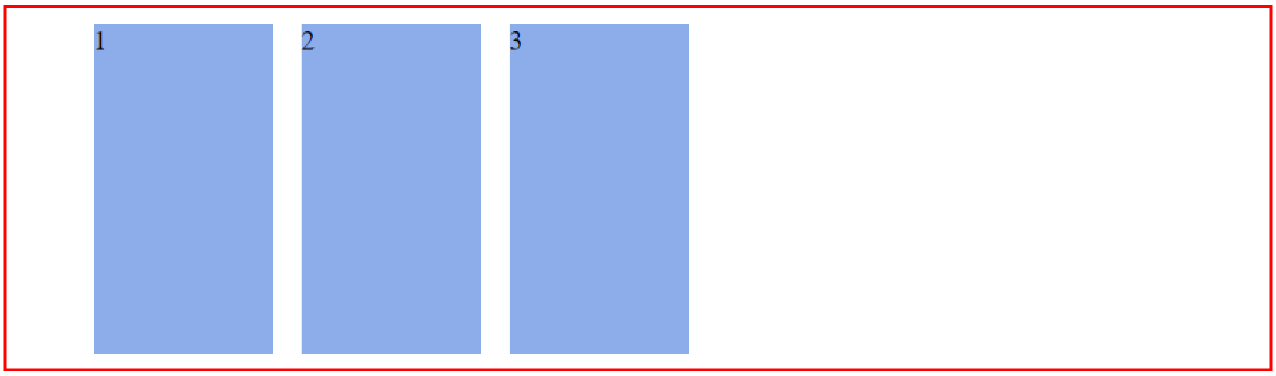
The different options for **align-items** are: **stretch**, **flex-start**, **flex-end**, **center** and **baseline**.

The **default** value is **stretch**. This will stretch the items to fill all available space along the cross axis. An element won't be stretched beyond its height or max-height.

Example

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <link rel="stylesheet" href="style.css">
        <title>CSS flexbox align items</title>
        <style>
            /*Make parent element a flex container*/
            ul {
            display: flex; /*or inline-flex*/

            align-items:stretch;
            border:2px solid red;
            height:200px;
            }
            li {
            width: 100px;
            background-color: #8cacea;
            margin: 8px;
            list-style-type:none;
            }
        </style>
    </head>
    <body>
    <ul> <!--parent element-->
        <li>1</li> <!--first child element-->
        <li>2</li> <!--second child element-->
        <li>3</li> <!--third child element-->
    </ul>
    </body>
</html>
```

The browser shows:

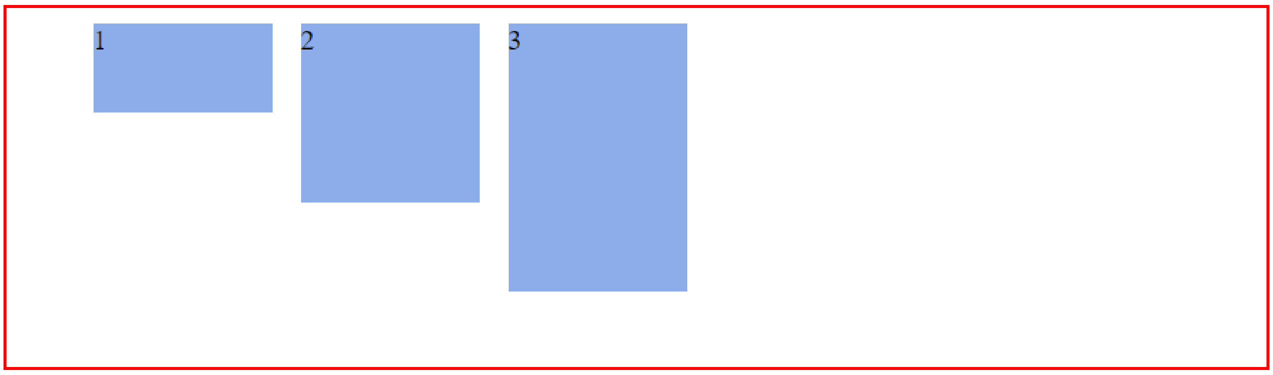


The **flex-start** groups the flex items to the start of the cross-axis.

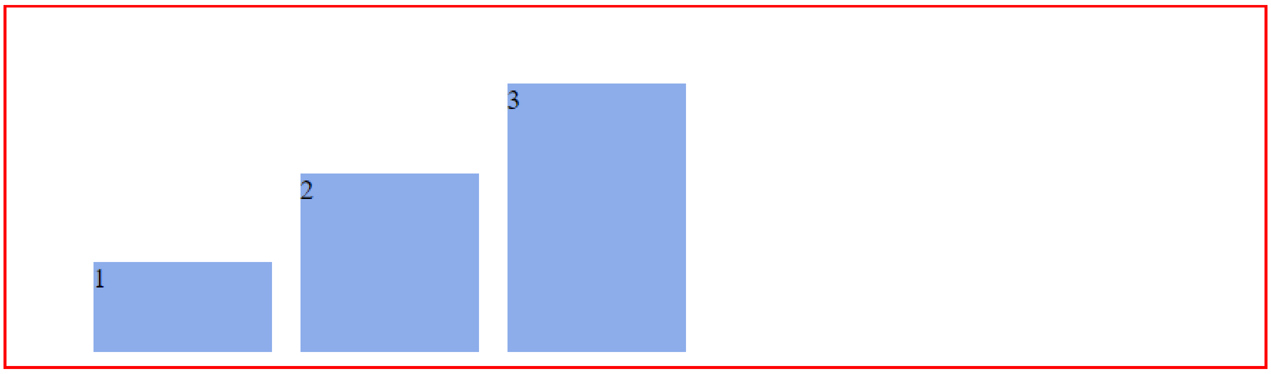
Example

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <link rel="stylesheet" href="style.css">
        <title>CSS flexbox align items</title>
        <style>
            /*Make parent element a flex container*/
            ul {
            display: flex; /*or inline-flex*/

            align-items:flex-start;
            border:2px solid red;
            height:200px;
            }
            li {
            width: 100px;
            background-color: #8cacea;
            margin: 8px;
            list-style-type:none;
            }
            li:nth-of-type(1){
                height:50px;
            }
            li:nth-of-type(2){
                height:100px;
            }
            li:nth-of-type(3){
                height:150px;
            }
        </style>
    </head>
    <body>
    <ul> <!--parent element-->
        <li>1</li> <!--first child element-->
        <li>2</li> <!--second child element-->
        <li>3</li> <!--third child element-->
    </ul>
    </body>
</html>
```

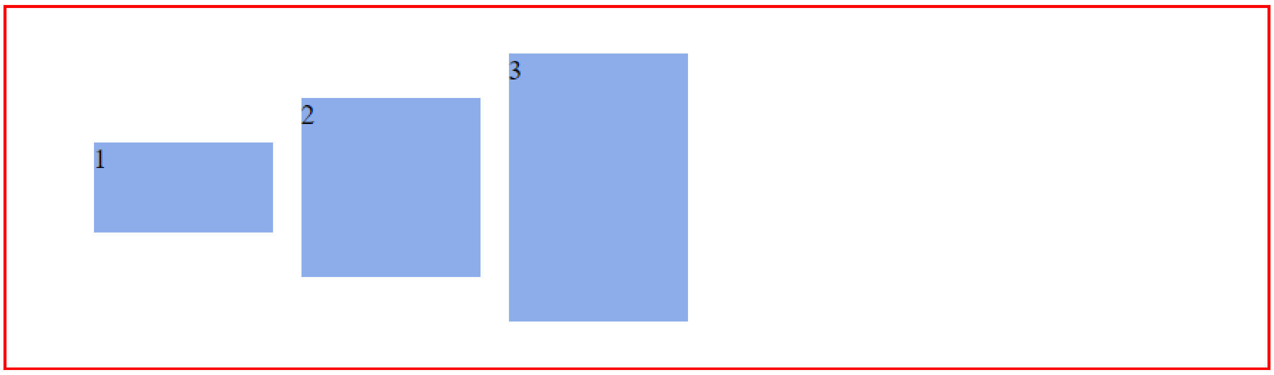
The browser shows:



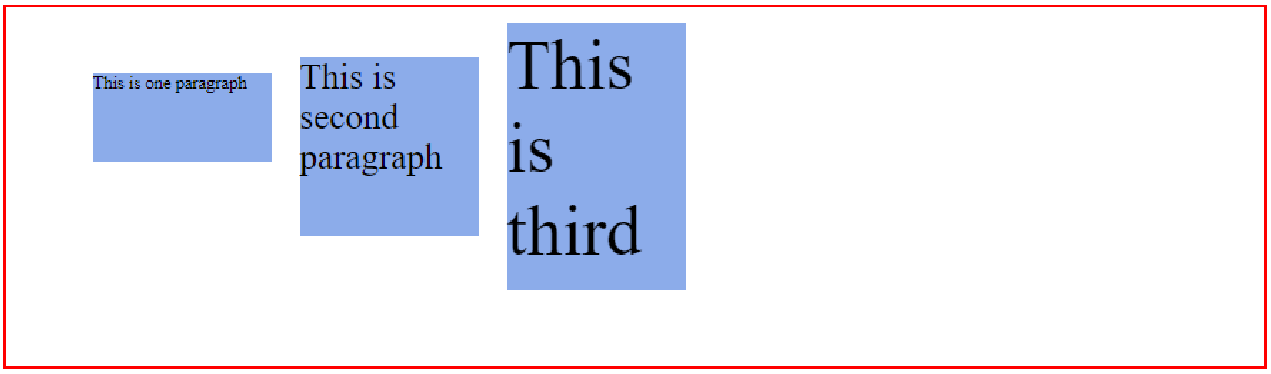
The **flex-end** groups the flex items to the end of the cross axis.



The **center** aligns the flex items to the center of the flex-container.



The **baseline** aligns items along the baseline of their text.



When there are multiple rows (or columns in a column flexbox layout), and there is extra space in the cross axis, **align-content** specifies how this space is distributed.

It takes the same values as align-items.

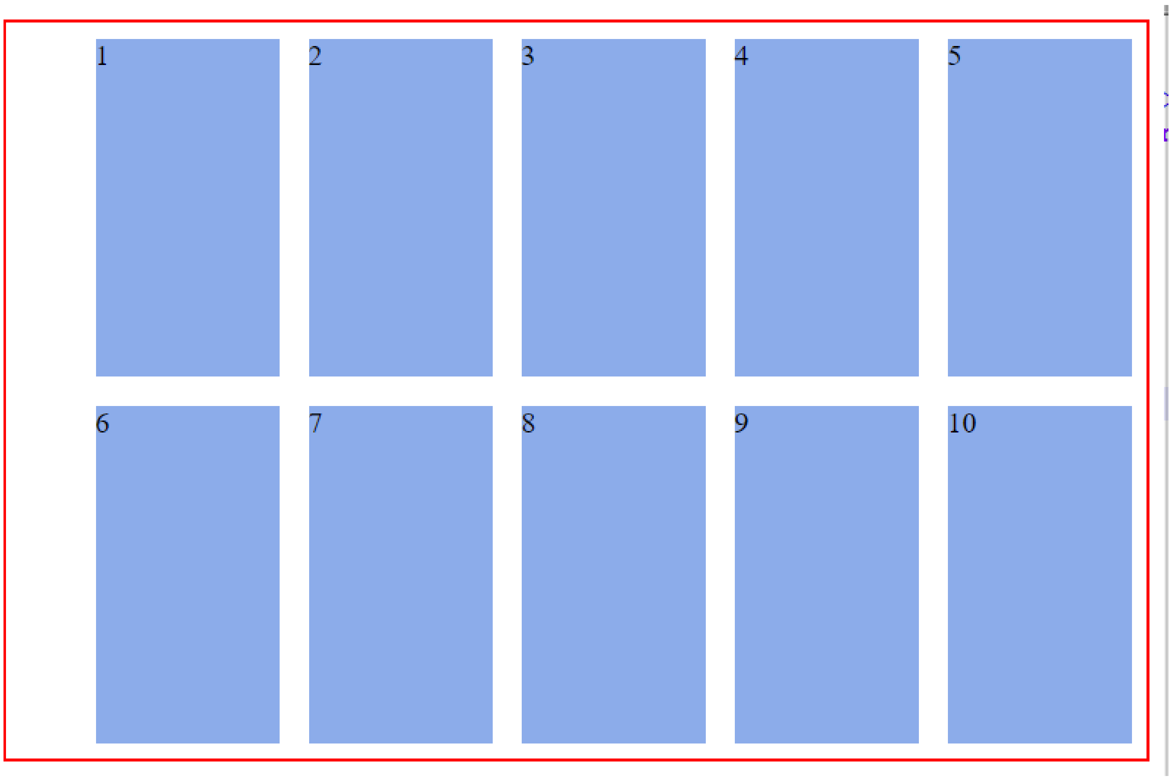
Just like align-items, the **default** value is also **stretch**.

With stretch the items stretches to fill all available space along the cross axis. An element won't be stretched beyond its height or max-height.

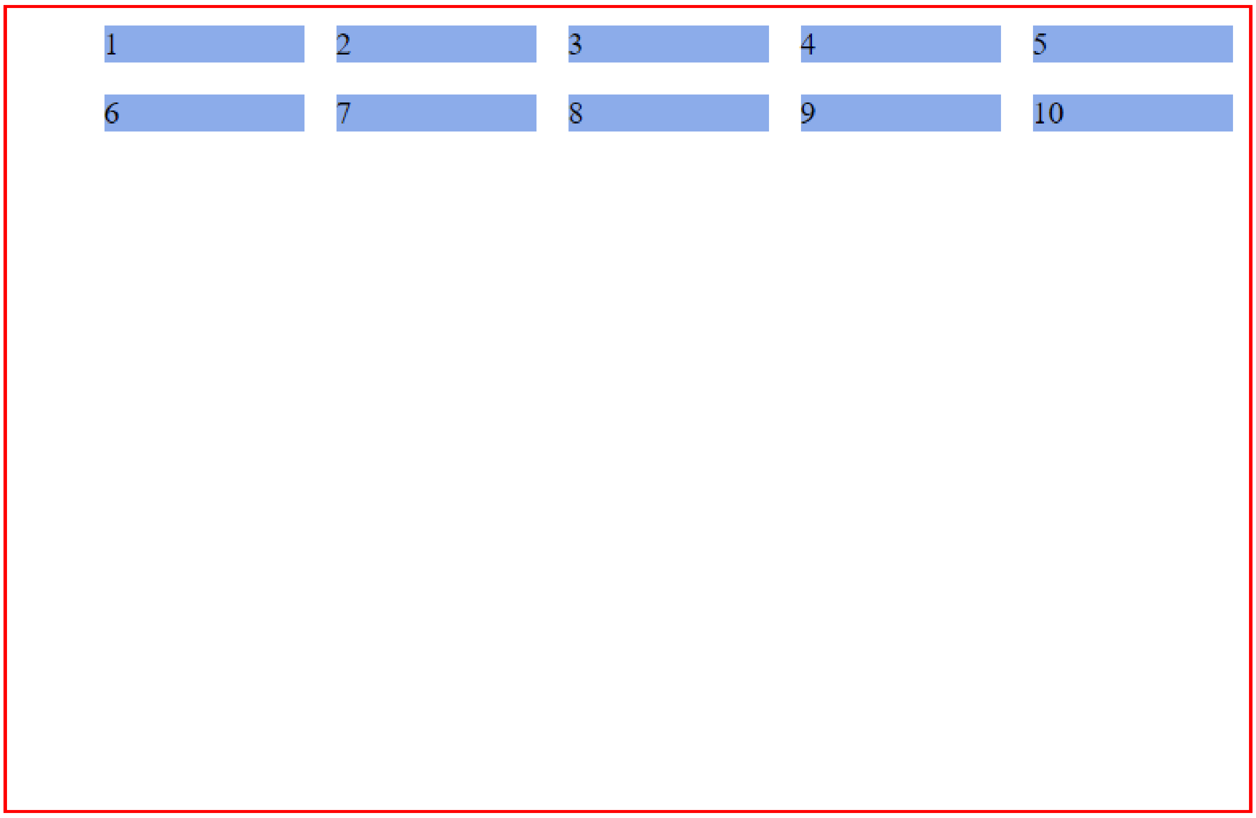
Example

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <link rel="stylesheet" href="style.css">
        <title>CSS flexbox align content</title>
        <style>
            /*Make parent element a flex container*/
            ul {
            display: flex; /*or inline-flex*/
            flex-wrap:wrap;
            align-content:stretch;
            border:2px solid red;
            height:400px;
            }
            li {
            width: 100px;
            background-color: #8cacea;
            margin: 8px;
            list-style-type:none;
            }

        </style>
    </head>
    <body>
    <ul> <!--parent element-->
        <li>1</li> <!--first child element-->
        <li>2</li> <!--second child element-->
        <li>3</li> <!--third child element-->
        <li>4</li>
        <li>5</li>
        <li>6</li>
        <li>7</li>
        <li>8</li>
        <li>9</li>
        <li>10</li>
    </ul>
    </body>
</html>
```

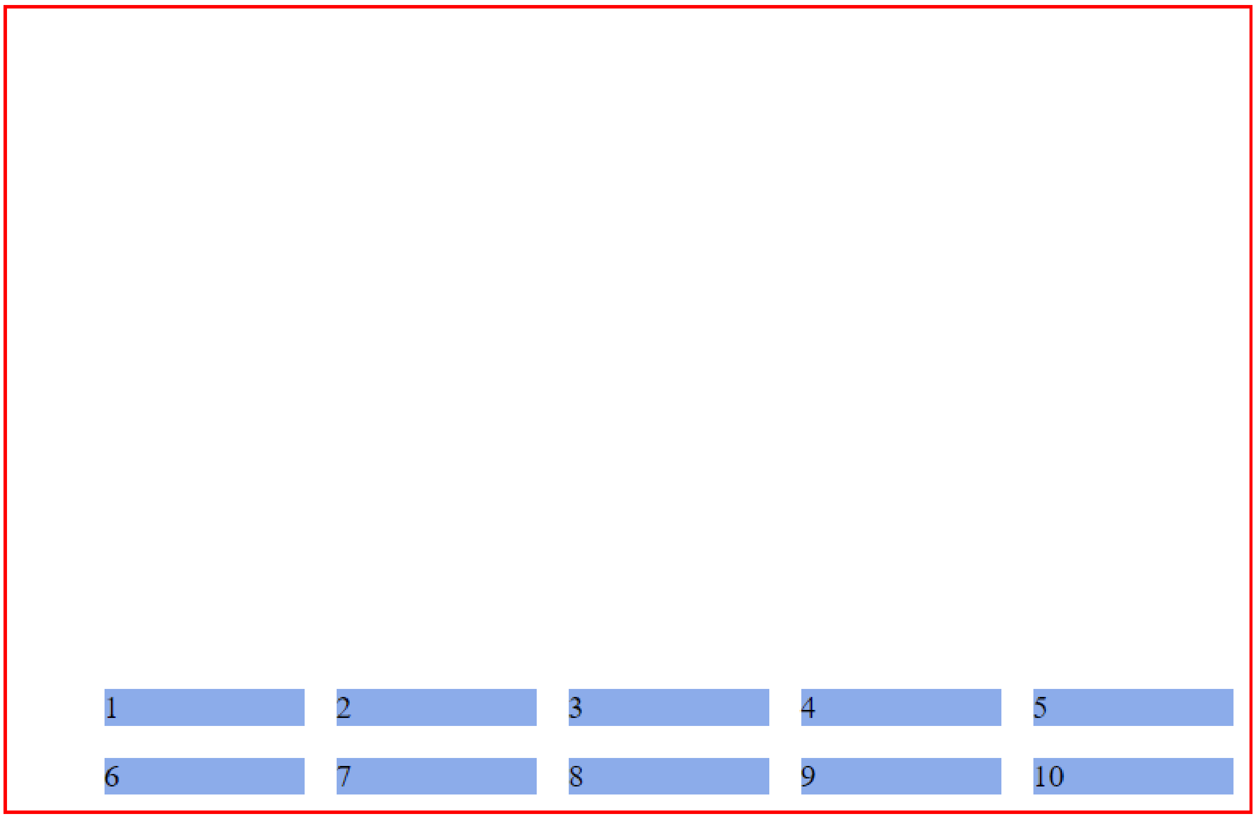
The browser shows:



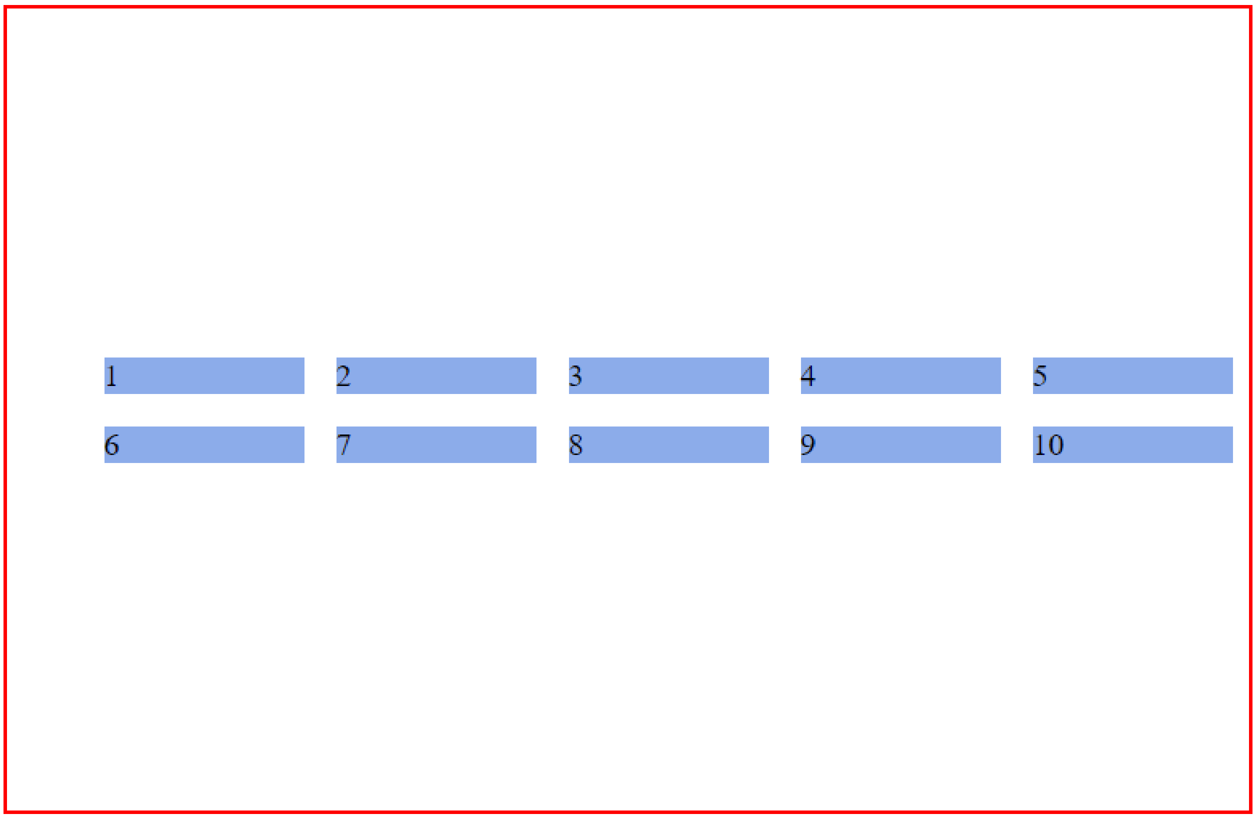
The value **flex-start** aligns the rows at the beginning of the cross axis.



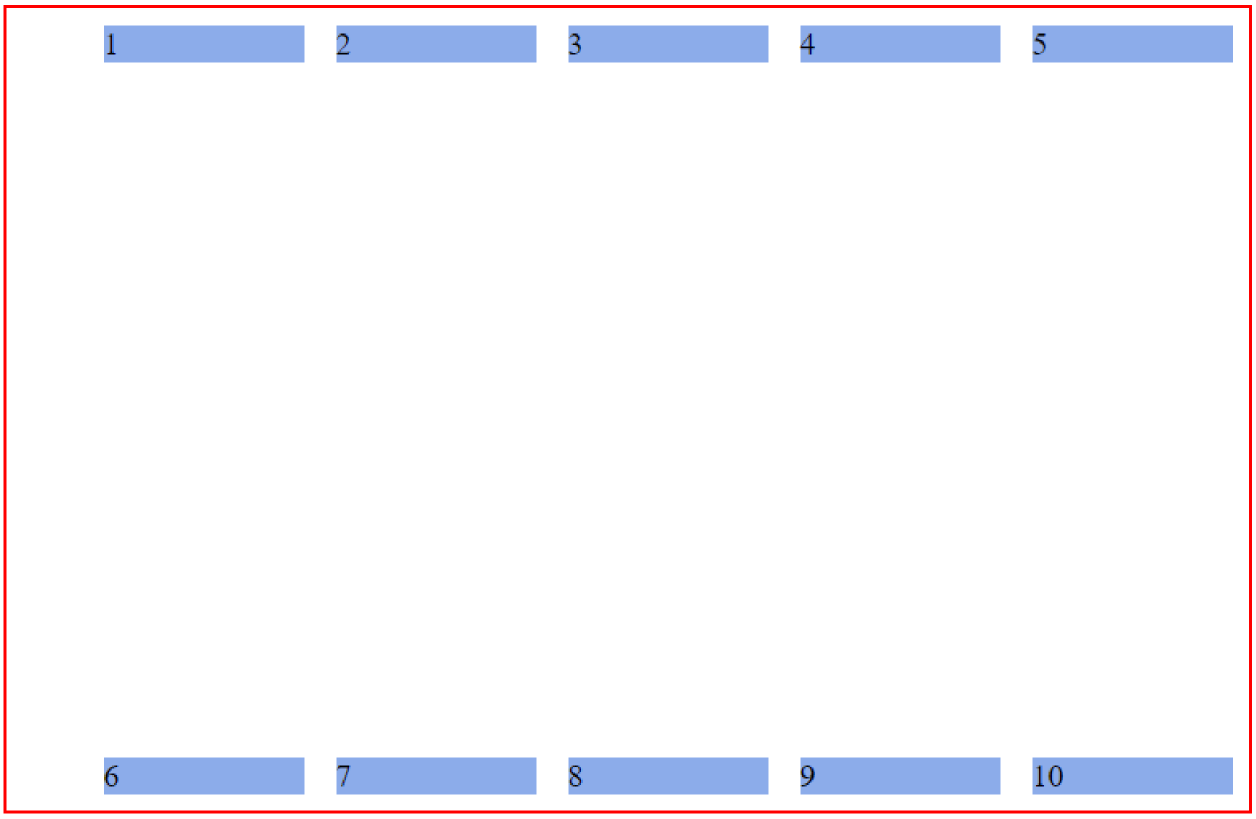
The value **flex-end** aligns the rows at the end of the cross axis.



The value **center** aligns the rows at the center of the cross axis.



The value **space-between** maximizes the spacing between the rows.



To have more control over **flex items** we can target them directly. We do this using four properties: **order**, **flex-grow**, **flex-shrink**, **flex-basis**.

The **order** property allows for reordering the flex items within a container.

Basically, with the order property you can move a flex-item from one position to another.

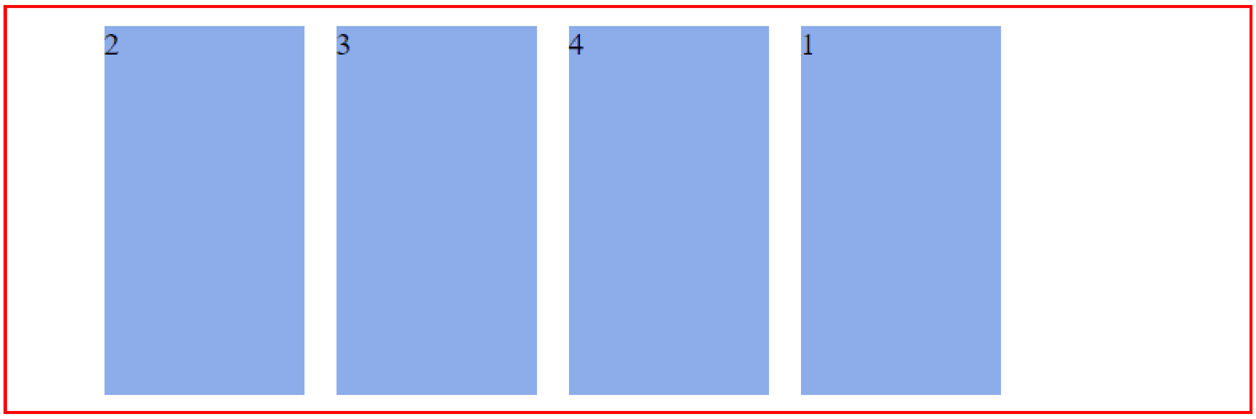
The **default value** for the **order** property is **0**. It may take on either negative or positive values.

The flex items are re-ordered based on the number values of the order property. From lowest to highest.

Example

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <link rel="stylesheet" href="style.css">
        <title>CSS flexbox order</title>
        <style>
            /*Make parent element a flex container*/
            ul {
            display: flex; /*or inline-flex*/
            border:2px solid red;
            height:200px;
            }
            li {
            width: 100px;
            background-color: #8cacea;
            margin: 8px;
            list-style-type:none;
            }
            li:nth-child(1){
                order:1;
            }
        </style>
    </head>
    <body>
    <ul> <!--parent element-->
        <li>1</li> <!--first child element-->
        <li>2</li> <!--second child element-->
        <li>3</li> <!--third child element-->
        <li>4</li>
    </ul>
    </body>
</html>
```

The items are reordered:



The **flex-grow** and **flex-shrink** properties control how much a flex-item should "grow" (extend) if there are extra spaces, or "shrink" if there are no extra spaces. They may take up any values ranging from 0 to any positive number.

Example

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <link rel="stylesheet" href="style.css">
        <title>CSS flexbox flex-grow</title>
        <style>
            /*Make parent element a flex container*/
            ul {
            display: flex; /*or inline-flex*/
            border:2px solid red;
            height:200px;
            }
            li {
            background-color: #8cacea;
            margin: 8px;
            list-style-type:none;
            flex-grow:1;
            }
        </style>
    </head>
    <body>
    <ul> <!--parent element-->
        <li>I'am a list item</li> <!--first child element-->
    </ul>
    </body>
</html>
```

The **flex-item** grows to occupy all the available space. If you tried resizing your browser, the flex-item would also shrink to accommodate the new screen width.

The shrink property is set to 1 by default.

The **flex-basis** property specifies the initial size of a flex-item. The default value if auto. Flex-basis can take on any values you'd use on the normal width property. That is, percentages, em, rem, pixel, etc.

When flex-basis property is set to auto, the width of the flex-item is computed automatically based on the content size. This means if you increase the content in the flex-item, it automatically resizes to fit.

If you want to set the flex-item to a fixed width, you can also do this:

li {flex-basis: 150px;}

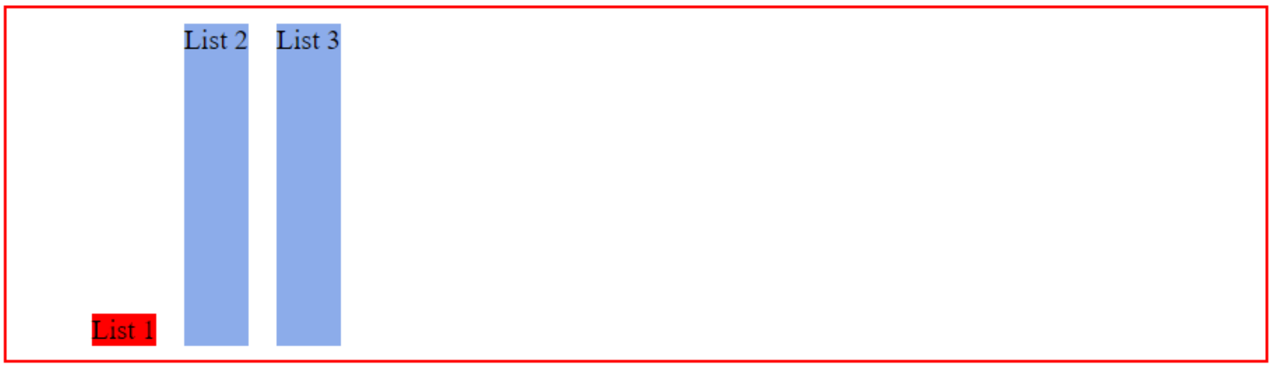
The **align-self** property takes a step further in giving us control over flex items.

If you want to change the position of a single flex-item along cross-axis without affecting the neighboring flex-items, you can use align-self. It may take on any of these values: auto, flex-start, flex-end, center, baseline, stretch.

Example

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <link rel="stylesheet" href="style.css">
        <title>CSS flexbox flex-grow</title>
        <style>
            /*Make parent element a flex container*/
            ul {
            display: flex; /*or inline-flex*/
            border:2px solid red;
            height:200px;
            }
            li {
            background-color: #8cacea;
            margin: 8px;
            list-style-type:none;
            }
            li:first-of-type{
                align-self:flex-end;
                background-color:red;
        </style>
    </head>
    <body>
    <ul> <!--parent element-->
        <li>List 1</li> <!--first child element-->
        <li>List 2</li>
        <li>List 3</li>
    </ul>
    </body>
</html>
```

The browser shows the following:



1.13 Media queries

The objective of media queries is to adapt the style sheet according to the device used to display the document: the type of media (screen, printer, etc.) and properties (dimensions, colours, …).

Think of media queries as if-else statements for the browser to interpret.

A media query is a boolean expression that allows us to specify the scope of CSS rules.

In the cascade mechanism, the filter by media is the first applied.

To make web pages responsive to smaller devices, media queries allow different style rules for different types of media in the same style sheet.

Example of what media query can do:

- When a user resizes the browser window up or down to a certain width or height, different layouts can be used by the browser from the media query.
- A small tablet can view a web page because a media query has been written to allow for the small display to view the page correctly.

A CSS media query within a style sheet can look like this:

```
@media only screen and (max-device-width: 600px) and (orientation: Landscape or Por-
trait){
 /*CSS rules on how to display with this resolution on this device*/
}
```

The preceding example tells the computer that if the current display has a width of 600 pixels, then run the code for that screen display.

The syntax is as follows:

```
@media [not | only] <media-type> [and] (<media-condition>);
```

Media Types in CSS: There are many types of media types which are listed below:

- **all**: It is used for all media devices
- **print**: It is used for printer
- **screen**: It is used for computer screens, smartphones, etc.
- **speech**: It is used for screen readers that read the screen aloud

Features of Media query: There are many features of media query which are listed below:

- **colour**: The number of bits per colour component for the output device.
- **grid**: Checks whether the device is grid or bitmap.
- **height**: The viewport height.
- **aspect ratio**: The ratio between width and height of the viewport.
- **color-index**: The number of colours the device can display.
- **max-resolution**: The maximum resolution of the device using dpi and dpcm.
- **monochrome**: The number of bits per color on a monochrome device.
- **scan**: The scanning of output devices.
- **update**: How quickly can the output device modify.
- **width**: The viewport width.

Example

```
@media only screen and (max-width: 500px) {
    body {
      background-color: black;
    }
}
```

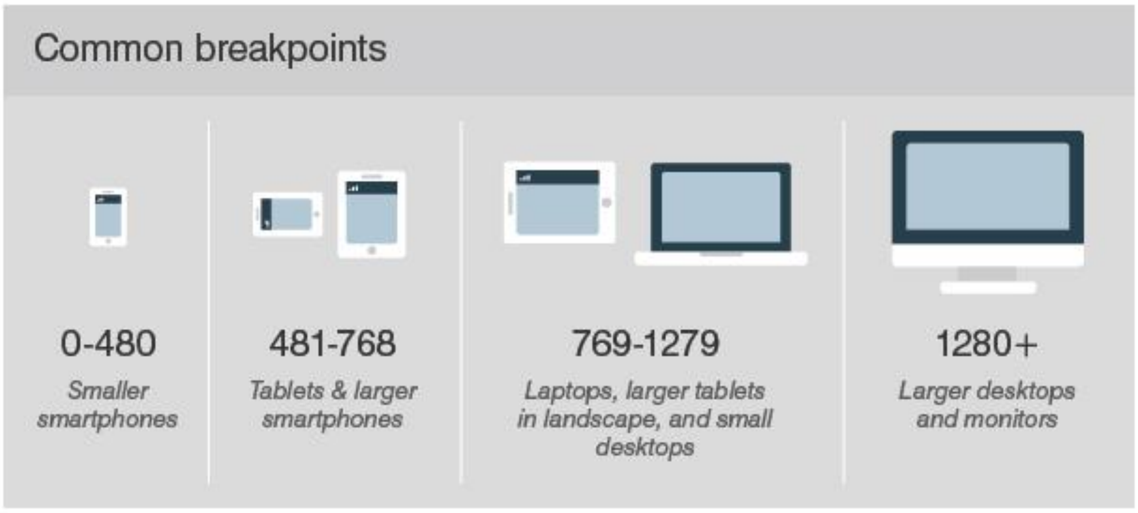
In the example, the CSS that sets the background colour to black is applied **if and only if** the width of the device is less than or equal to 500px.

Example

```
@media only screen and (min-width: 300px) and (max-width: 500px) {
  body {
    background-color: black;
  }
}
```

Here, we are saying to execute the code if and only if the minimum size of the browser is 300px, and the maximum is 500px.

Breakpoints are browser widths at which your site's content will respond to provide users with the best possible layout. Whereas you can use standard breakpoints when designing responsive pages, it is recommended creating these based on content – never on specific devices – to ensure a better user experience. It is best practices to design for your smallest viewport first: design the content to fit on a small screen size first and then expand the screen until a breakpoint becomes necessary.


Common breakpoints
0-480
Smaller smartphones
481-768
Tablets & larger smartphones
769-1279
Laptops, larger tablets in landscape, and small desktops
1280+
Larger desktops and monitors

It is possible to link different stylesheets for different screen sizes. The way to do that is to use the media attribute of the link tag.

```
<link rel="stylesheet" type="text/css" media="only screen and (max-device-width:
480px)" href="mobile-device.css">
```

Now you can define all the mobile specific styles in the mobile-devices.css file.