Styling Maps Using Advanced CartoCSS Techniques

The previous reference covers the basics of CartoCSS: its syntax and common properties that you will encounter while using CartoCSS. This reference covers more advanced techniques.

Combining selectors

You can **combine conditional selectors** by putting them directly next to each other:

```
#layer-name[attr1 = value1][attr2 > value2] { ... }
```

This statement will only apply to features where **all** conditions are true. You can combine as many conditions as needed in this way. For example, to style only those buildings in New York that are over 50 feet tall, you might write:

```
#buildings[state = 'New York'][height > 50] { ... }
```

If you find yourself writing things like this to apply styles when one condition or the other is true (`attr1 = value1 OR attr2 > value2`):

```
#layer-name[attr1 = value1] {
     property: value;
}
#layer-name[attr2 > value2] {
     property: value;
}
```

consider separating the selectors with a comma:

```
#layer-name[attr1 = value1],
#layer-name[attr2 > value2] {
     property: value;
}
```

This does the same thing, but you don't have to repeat the styles (`property: value`) and if you have to change it later it will be faster.

Finally, you can **nest statements**. This says the same thing as the statement above:

```
#layer-name {
     [attr1 = value1],
     [attr2 > value2] {
```

```
            property: value;
        }
    }
```

Let's make this more concrete:

```
    #buildings {
        [state = 'New York'],
        [height > 50] {
            marker-fill: red;
        }
    }
```

This styles features in the `buildings` layer that either have `state` set to New York or `height` greater than 50 such that their marker fill is red.

Variables

Sometimes you will find yourself repeating values in your statements. Your statements can be made more flexible using variables. Creating a variable looks like this:

```
    @variable: value;
```

for example:

```
    @roadcolor: #ff307a;
```

Then, instead of using the value in your statements, use `@variable`. For example:

```
    #roads {
        line-color: @roadcolor;
    }
```

You can use math on variables. For example, if you have a variable named `@width` for the width of your markers and want to double the marker width at zoom level 14, you could do something like this:

```
    #buildings[zoom >= 14] {
        marker-width: @width * 2;
    }
```

In the same way, you can use all basic arithmetic (`*`, `/`, `+`, `-`) on variables.

Color functions

Similarly to arithmetic with numbers, you can use functions on colors to change them. A full list of these functions is available in the official documentation for CartoCSS, but we will list a few here:

```
lighten()
darken()
fadein()
fadeout()
```

These each take a color and a percentage (the amount you want to lighten, darken, etc, the color). For example, if you have a variable called `@buildingcolor`, you might want to fill the marker and outline the color two different but related colors:

```
#buildings {
     marker-fill: @buildingcolor;
     marker-line-color: lighten(@buildingcolor, 25%);
}
```

This makes the markers have a fill of `@buildingcolor`, and the outlines of the markers are the same color but 25% lighter.

You don't have to use a variable for the color, but it is a common way to use these functions.

Using attributes for property values

Sometimes you want properties to get values based on the features' attributes but without splitting the features into classes. For example, instead of styling using ranges:

```
#buildings[height > 50][height <= 100] {
     marker-width: 5;
}
```

We can write code that makes the markers' width always one-tenth of the building's height:

```
# buildings {
     marker-width: [height] / 10;
}
```

That is, we surround the attribute's name in square brackets `[]` and can perform arithmetic on the attribute values as we would with a variable. When appropriate, this is useful as it reduces the amount of code we have to write.