# Styling R plots with cascading style sheets and Rcssplot

Tomasz Konopka

December 13, 2019

**Abstract**

Package Rcssplot provides a framework for customizing R plots in a way that separates data-handling code from appearance-determining settings.

## 1 Introduction

The R environment provides provides numerous ways to fine-tune the appearance of plots and charts. Taking advantage of these features can make complex data visualizations more appealing and meaningful. For example, customization can make some components in a composite visualization stand out from the background. However, such tuning can result in code that is long and complex.

A specific problem with code for graphics is that it often mixes operations on data with book-keeping of visual appearance. The mixture makes such code difficult to maintain and extend. A similar problem in web develpment is addressed by separating style settings from content using cascading style sheets. The Rcssplot package implements a similar mechanism for the R environment.

This vignette is organized as follows. The next section reviews how to create composite visualizations with base graphics. Later sections describe how to manage visual style using Rcssplot. The vignette ends with a summary and pointers to other graphics frameworks and packages.

## 2 Styling plots with base graphics

To start, let's look at styling plots using R's built-in capabilities, called 'base graphics'. For concreteness, let's use an example with a bar chart and a small data vector.

```
a <- setNames(c(35, 55, 65, 75, 80, 80), letters[1:6])
a

##  a  b  c  d  e  f
## 35 55 65 75 80 80
```

The function to draw a bar chart in R is `barplot`. We can apply it on this data, `a`, to obtain a chart with R's default visual style (Figure 1A).

```
barplot(a, main="Base graphics")
```

The output has many of the elements that we expect from a bar chart (bars, axes, etc.). But there is room for improvement. At a minimum, the chart requires a title and a label for the vertical axis. We might also like to change some colors and spacings. Many of these features can be tuned directly through the `barplot` function (Figure 1B).

```
barplot(a, main="Manual tuning", ylab="y label", col="#000080", border=NA, space=0.35)
```

The function call is now longer, but the output is more complete.

It is possible to tune the plot further using other arguments to `barplot`. However, some aspects of the chart, for example margins, are not accessible in this manner. Furthermore, we may wish to add other custom elements to the chart area, for example a subtitle. To adjust or to create these elements, it is necessary to issue several function calls. In practice it is convenient to encapsulate such commands in a custom function.

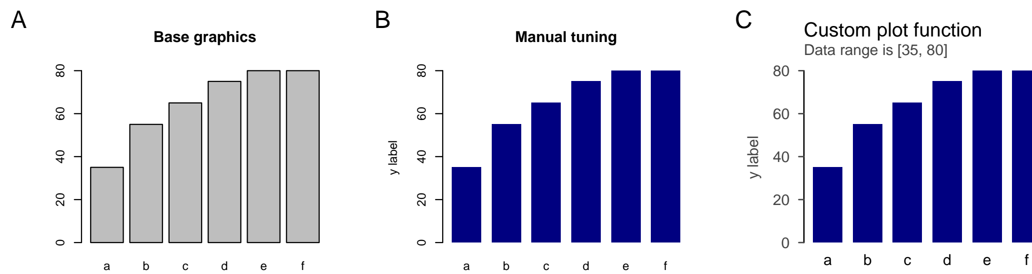


Figure 1: Charts created with base graphics using: (A) R's `barplot` function and default settings; (B) R's `barplot` function and custom settings; (C) a custom plot function that styles bars, axes, and labels individually.

```
## helper builds a string describing a range
range.string <- function(x) {
    paste0("Data range is [", min(x), ", ", max(x), "]")
}
## barplot with several custom settings and components
base.barplot.1 <- function(x, main="Custom plot function", ylab="y label") {
  ## start with a plot with bars, but nothing else
  barpos <- barplot(x, col="#000080", axes=FALSE, axisnames=FALSE,
                    border=NA, space=0.35)
  ## add custom components
  axis(1, at=barpos[,1], labels=names(x), lwd=0, col="#111111", cex.axis=1.2,
       line=-0.35)
  axis(2, col.ticks="#444444", col.axis="#444444", cex.axis=1.2, lwd=1.2, las=1,
       tck=-0.03, lwd.ticks=1.2)
  mtext(main, adj=0, line=2.2, cex=1.1)
  mtext(range.string(x), adj=0, line=0.9, cex=0.8, col="#444444")
  mtext(ylab, side=2, cex=0.8, line=3, col="#444444")
}
```

The first block above is a helper function to construct a subtitle. The second block is a definition of function `base.barplot.1`. It takes as input a data vector and two strings for the title and y-axis label. The first line of the function body creates a chart without excess decorations. Subsequent lines add axes and labels. Each command carries several custom settings.

We can now apply the custom function on our data (Figure 1C).

```
base.barplot.1(a)
```

The function call is concise, yet its output is a bar chart that looks legible and sophisticated.

Coding custom functions like `base.barplot.1` is the usual way for making composite charts with R's base graphics. However, this approach has some disadvantages.

- The custom function is now so specialized that it may only be fit for one-time use. We cannot easily change any visual aspects without updating the function definition.
- Because the function mixes code that manipulates data with code that adjusts visual appearance, there are opportunities to introduce bugs during tuning or maintenance.
- It is rather difficult to create a second function with the same visual style and to keep these styles consistent throughout the lifetime of a long project.

These observations stem from the fact that the custom function performs several distinct roles. First, it combines graphical commands to create a composite visualization. Second, it performs some useful manipulations on the data (here, compute the range). Third, the function styles graphical components. The difficulties in maintenance all arise from the styling role. Thus, it would be useful to separate this role from the others, i.e. to provide styling settings that are independent from the data-handling instructions.

## 3 Styling with cascading style sheets

The Rcssplot package provides a mechanism to style R's graphics that is inspired by cascading style sheets (css) used in web-page design. In this approach, settings for visual representation are stored in a file that is separate from both the raw data and the code that creates visualizations.

To use this framework, we load the package.

```
library(Rcssplot)
```

This command triggers some messages from the R environment. They convey that the package provides new implementations for graphics functions. This means that executing commands with one of the listed names, for example `mtext`, will launch implementations provided by Rcssplot rather than by the core packages `graphics` and `grDevices`. The new implementations are designed to mimic behaviors of the familiar base graphics, but also add new features. (Compatibility with base graphics is discussed further in one of the appendices).

### 3.1 Rcss styles

Let's adopt a convention whereby files with style definitions have `Rcss` extensions. As an exampe, a file called `vignettes.bar1.Rcss` is available in a sub-folder along with the package vignette. The file is formatted in a similar way to cascading style sheets, `css`, that are ubiquitous in web design.

```
barplot {
  border: NA;
  col: #000080;
  space: 0.35;
}
```

The content is a block with the name `barplot`. This corresponds to R's function for bar charts. Elements within the block are property/value pairs that correspond to the function arguments.

We can read this style definition into the R environment using function `Rcss`.

```
style1 <- Rcss(file.path("Rcss", "vignettes.bar1.Rcss"))
```

We can look inside the object to check that it loaded correctly.

```
style1

## Rcssplot:
## Defined selectors:  barplot
## Use function printRcss() to view details for individual selectors

printRcss(style1, "barplot")

## Rcssplot: barplot
##     | border: NA
##     | col: #000080
##     | space: 0.35
##
##     Defined classes:
```

The first command displays some basic information about the style. The second command shows more details for the `barplot` component (called a selector). In this case, we recognize the three property/value pairs from the `Rcss` file.

Next, let's use the style object in a plot. The Rcssplot package provides functions that mask many of R's graphics functions. These new function replicate the base functions, but also add some new features. In practice, familiar graphics commands can be used as before, or with additional arguments. For example, we can create a simple barplot (Figure 2A).

```
barplot(a, main="Rcssbarplot, unstyled", ylab="y label")
```

In this plain form, the syntax as well as the output is exactly the same as using base graphics `barplot`. But we can add styling by passing a style object as an argument (Figure 2B).

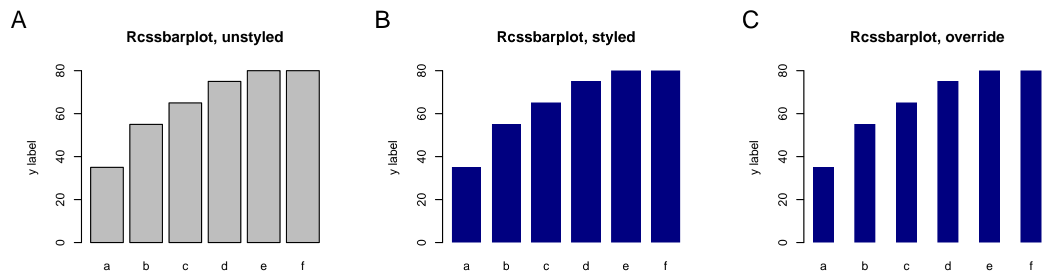


Figure 2: Charts created with Rcssplot using: (A) the default style; (B) a style determined through a style sheet; (C) a style sheet, but with the bar width over-ridden by a setting within a function call.

```
barplot(a, main="Rcssbarplot, styled", ylab="y label", Rcss=style1)
```

The output is analogous to one of the previous examples (c.f. Figure 1B). Previously, we achieved the effect by specifying three arguments (`border`, `col`, and `space`). This alternative requires only one argument: custom settings are extracted automatically from the style object, `style1`.

In some cases it is useful to override settings defined in a style sheet (Figure 2C).

```
barplot(a, main="Rcssbarplot, override", ylab="y label", space=1, Rcss=style1)
```

Here, the bar width is determined by `space=1` in the function call despite this property being also specified in the style object. Thus, values set manually take precedence over cascading style sheets.

### 3.2 Rcss classes

Next, let's implement the entire custom bar plot using style sheets and introduce a new feature - style classes. We need additional css definitions encoded in another file, `vignettes.bar2.Rcss`.

```
axis {
  cex.axis: 1.2;
}
axis.x {
  line: -0.35;
  lwd: 0;
}
mtext.ylab, mtext.submain, axis.y {
  col: #444444;
}
axis.y {
  col.axis: #444444;
  col.ticks: #444444;
  las: 1;
  lwd: 1.2;
  lwd.ticks: 1.2;
  tck: -0.03;
}
mtext {
  cex: 0.8;
  adj: 0;
}
mtext.main {
  line: 2.2;
  cex: 1.1;
}
mtext.ylab {
  line: 3;
  adj: 0.5;
}
mtext.submain {
```

```
  line: 0.9;
}
```

The definitions are again arranged into blocks that correspond to R's base graphics commands.

- The values in the style sheet match the settings hard-coded into function `base.barplot.1`. The format of the style sheet makes it easy to identify property/value pairs.
- Some blocks contain names with dots followed by a string, e.g. `axis.x`. This notation defines property/value pairs that are activated only in particular circumstances. In the case of `axis.x`, the definitions pertain to function `axis`, but only when accompanied by class label `x`. This will become clearer below.
- Some blocks contain names for several base graphics components separated by commas, e.g. `mtext.ylab, mtext.submain, axis.y`. This syntax defines property/value pairs for several components at once. In this case, it is convenient to specify a common color.

We can now write a new function that can apply these styles.

```
## barplot using Rcssplot, version 1
rcss.barplot.1 <- function(x, main="Custom Rcss plot", ylab="y label",
                           Rcss="default", Rcssclass=c()) {
  ## create an empty barplot
  barpos <- barplot(x, axes=FALSE, axisnames=FALSE, Rcss=Rcss, Rcssclass=Rcssclass)
  ## add custom components
  axis(1, at=barpos[,1], labels=names(x), Rcss=Rcss, Rcssclass=c(Rcssclass,"x"))
  axis(2, Rcss=Rcss, Rcssclass=c(Rcssclass,"y"))
  mtext(main, Rcss=Rcss, Rcssclass=c(Rcssclass,"main"))
  mtext(range.string(x), Rcss=Rcss, Rcssclass=c(Rcssclass, "submain"))
  mtext(ylab, side=2, Rcss=Rcss, Rcssclass=c(Rcssclass,"ylab"))
}
```

The structure mirrors `base.barplot.1`, but also accepts an `Rcss` object and a vector `Rcssclass`. Within the function body, all the custom graphical settings are replaced by an `Rcss` argument and a vector for `Rcssclass`. When there are multiple calls to one graphic function (e.g. `axis` for the x and y axes), the `Rcssclass` vector contains some distinguishing labels. These labels match the css subclasses we saw previously.

The output from the new function is a complete plot with all our custom settings (Figure 3A).

```
style2 <- Rcss(file.path("Rcss", c("vignettes.bar1.Rcss", "vignettes.bar2.Rcss")))
rcss.barplot.1(a, main="Rcss style2", Rcss=style2)
```

The first line creates a new style object, `style2`, using the `Rcss` definitions from both files displayed above. The call to `rcss.barplot.1` then creates the chart.

The advantage of this approach is that we can now change the visual output by replacing the `Rcss` style object by another one without re-coding the custom function. One way to change the style is to edit the `Rcss` files (or use differet files), load the definitions into a new style object, and generate a new figure with the new style. Another way, covered next, is to define multiple styles within one `Rcss` object.

## 3.3 Multiple styles

Let's look at another `Rcss` file, `vignettes.bar3.Rcss`.

```
barplot.typeB {
  col: #449944;
  space: 0.6;
}
mtext.typeB.main {
  cex: 1.0;
  font: 2;
}
```

The two blocks are decorated with a subclass called `typeB`. This class name is not explicitly used within the code of the plot function `rcss.barplot.1`. However, we can prime the plot function to use these definitions by providing the class name during the function call (Figure 3B).

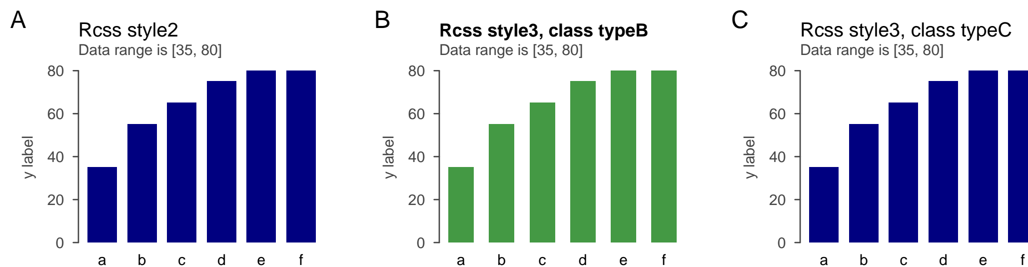


Figure 3: Charts created by custom plot functions with base graphics and Rcssplot using: (A) a style determined by css; (B) a style sub-class defined in css; (C) a style sub-class that is not defined in css (equivalent to (A)).

```
style3 <- Rcss(paste0("Rcss/vignettes.bar", c(1, 2, 3), ".Rcss"))
rcss.barplot.1(a, main="Rcss style3, class typeB", Rcss=style3, Rcssclass="typeB")
```

The output now incorporates settings defined in the generic `barplot` and `mtext` css blocks, but also those settings targeted using the `typeB` subclass. As in conventional cascading style sheets, when a parameter is specified in multiple locations with an `Rcss` object, the definition with the more specific class takes precedence.

When the `Rcssclass` argument contains items that are not recognized, these items are just ignored (Figure 3C).

```
rcss.barplot.1(a, main="Rcss style3, class typeC", Rcss=style3, Rcssclass="typeC")
```

Here, the class name `typeC` does not appear in the underlying style sheet files, so the output is the same as if this subclass was not specified at all.

In summary, we saw in this section how to use cascading style sheets to determine visual appearance. This approach has several advantages over using base graphics alone.

- The new function separates the details of visualization from the R code. This makes it easier to tweak aesthetics (in the `Rcss` files) without worrying about the code structure.
- The new function is shorter because calls to commands that generate structure (e.g. `axis` and `mtext`) are not interspersed with details of graphical parameters. This makes it easier to see the organization of the composite graphic.
- The styles can be reused in several custom functions. Thus, it is straightforward to maintain a uniform style across a family of functions.

In the next section we will look at additional features that simplify creation of custom graphics.

# 4 Additional features

This section covers some additional features provided by the package. The first two subsections deal with reducing repetitive code. The third subsection introduces usage of css objects as general data stores. Finally, the fourth subsection introduces a system to simplify development through trials and adjustments.

## 4.1 Default styles and compulsory classes

While the code in `rcss.barplot.1` is simpler than in `base.barplot.1`, it still contains repetitive elements. In particualr, constructions `Rcss=Rcss` and `Rcssclass=Rcssclass` appear in almost every line. We can avoid this repetition by setting a default style and a compulsory class through `RcssDefaultStyle` and `RcssCompulsoryClass`. These objects can be defined in any environment, for example inside a function. Consider the following adjustment of our barplot function.

```
## barplot using Rcssplot, version 2 (using a default style and a compulsory class)
rcss.barplot.2 <- function(x, main="Custom Rcss plot", ylab="y label",
                           Rcss="default", Rcssclass=c()) {
  RcssDefaultStyle <- RcssGetDefaultStyle(Rcss)
```

```
  RcssCompulsoryClass <- RcssGetCompulsoryClass(Rcssclass)
  ## create a barplot (without Rcss arguments)
  barpos <- barplot(x, axes=FALSE, axisnames=FALSE)
  axis(1, at=barpos[,1], labels=names(x), Rcssclass="x")
  axis(2, Rcssclass="y")
  mtext(main, Rcssclass="main")
  mtext(range.string(x), Rcssclass="submain")
  mtext(ylab, side=2, Rcssclass="ylab")
}
```

The preparation steps set a default style and compulsory class. Subsequent calls to graphics functions do not to refer to objects `Rcss` or `Rcssclass`. Nonetheless, the output of the custom function can exhibit styling.

- Calls to `axis` and `mtext` in the above function still carry `Rcssclass` arguments. These are necessary to distinguish styling between the x- and y-axis, and between the title and sub-title. However, setting the compulsory class reduces clutter (no need to write `Rcssclass=Rcssclass`).
- It is important that the preparation steps set `RcssDefaultStyle` and `RcssCompulsoryClass` with the help of function calls. Their role will become more clear in the next section. In short, those functions help preserve defaults that may have been set outside of the custom function.

## 4.2 Global defaults

In the previous example, `rcss.barplot.2`, we changed the default style within the custom function, i.e. in a local environment. It is also possible to apply such changes in the global environment.

```
RcssDefaultStyle <- style3
RcssCompulsoryClass <- c()
```

When the styling is set as above, i.e. outside a function definition, the custom barplot function can be simplified even further.

```
## barplot using Rcssplot, version 3 (assumes global style)
rcss.barplot.3 <- function(x, main="Custom Rcss plot", ylab="y label",
                           Rcssclass="typeB") {
  ## adjust compulsory class
  RcssCompulsoryClass <- RcssGetCompulsoryClass(Rcssclass)
  ## create a barplot
  barpos <- barplot(x, axes=FALSE, axisnames=FALSE)
  axis(1, at=barpos[,1], labels=names(x), Rcssclass="x")
  axis(2, Rcssclass="y")
  mtext(main, Rcssclass="main")
  mtext(range.string(x), Rcssclass="submain")
  mtext(ylab, side=2, Rcssclass="ylab")
}
```

- The function definition no longer carries an argument `Rcss`. The style is assumed to come entirely from the default style.
- The function still carries an argument `Rcssclass`. Keeping this argument is a mechanism to use sub-classes without the need to repeatedly redefine the compulsory class in the global environment.

Sometimes, we may want to reset the default style and/or the compulsory style class(es). This can be achieved by setting those objects to `NULL`.

```
RcssDefaultStyle <- NULL
RcssCompulsoryClass <- NULL
```

Now that we've adjusted default settings within custom functions as well as in the global environment, let's revisit revisit the functions `RcssGetDefaultStyle` and `RcssGetCompulsoryClass`. Consider the following snippet.

```
RcssCompulsoryClass <- "bar0"
RcssCompulsoryClass
## [1] "bar0"
foo1 <- function() {
    RcssCompulsoryClass <- "bar1"
    RcssCompulsoryClass
}
foo1()
## [1] "bar1"
```

The first result is `bar0`; let's think of this as a css class that we wish to employ at a global level. In the first function, `foo1`, the compulsory class is set with a naive assignment. The return value reveals that within that function, the compulsory class becomes `bar1` and our previous value `bar0` is lost. This is normal behavior, but it does not reflect our intention to keep `bar0` as a global style class.

To keep the intended global class, we can use function `RcssGetCompulsoryClass`.

```
foo2 <- function() {
   RcssCompulsoryClass <- RcssGetCompulsoryClass("bar2")
   RcssCompulsoryClass
}
foo2()
## [1] "bar0" "bar2"
RcssCompulsoryClass
## [1] "bar0"
```

Here, `foo2` looks up the compulsory class set in parent environments and augments it with the new label. The effective compulsory class within that function thus becomes a combination of the global and local settings. The final command show that `RcssCompulsoryClass` in the global environment remains unaffected. Labels `bar1` and `bar2` are thus localized to the custom functions.

The function `RcssGetDefaultStyle` fulfills an analogous role for style objects. Using a function call `RcssGetDefaultStyle("default")` returns an object equivalent to the one set in a parent environment.

### 4.3 Custom selectors

We've already seen that Rcss files can store settings for familiar graphics settings. But cascading style sheets can also be used to encode other settings as well, indeed any property/value pairs. Consider style file `vignettes.bar4.Rcss`.

```
baraxis {
  stripe: 1;
}
barplot.dotted {
  col: #9999cc;
}
baraxis.dotted {
  stripe: 1;
  ylim: 0 101;
}
abline.dotted {
  col: #666666;
  lty: 2;
}
```

The first block is named `baraxis`, but this does not correspond to any of R's base graphics commands. Therefore, this block does not have any direct effect on styling. But we can write code to exploit information in `baraxis` by extracting values manually. There are two functions for this purpose, `RcssProperty` and `RcssValue`.

```r
style4 <- Rcss(file.path("Rcss", paste0("vignettes.bar", c(1, 2, 4), ".Rcss")))
RcssProperty("baraxis", "stripe", Rcss=style4)
```

```
## $defined
## [1] TRUE
##
## $value
## [1] 1
```

The output signals that the `stripe` property in a `baraxis` block is indeed defined, and provides its value. A related command automatically substitutes undefined values with a default.

```r
RcssValue("baraxis", "stripe", default=0, Rcss=style4)
```

```
## [1] 1
```

```r
RcssValue("baraxis", "strpe", default=0, Rcss=style4)
```

```
## [1] 0
```

The result here is `1` for `stripe` because we saw this property is defined; the suggestion `default=0` is ignored. The second result is `0` because the misspelling `strpe` is not present in the file.

We can now exploit this feature to augment our bar chart with an option to draw horizontal rules instead of a y-axis.

```r
## barplot using Rcssplot, version 5 (uses custom css selectors)
rcss.barplot.4 <- function(x, main="", ylab="Proportion (%)",
                           Rcss="default", Rcssclass=c()) {
  ## use custom style, compulsary class
  RcssDefaultStyle <- RcssGetDefaultStyle(Rcss)
  RcssCompulsoryClass <- RcssGetCompulsoryClass(Rcssclass)
  ## extract custom properties - show axis? force ylim?
  stripes <- RcssValue("baraxis", "stripe", default=0)
  ylim <- RcssValue("baraxis", "ylim", default=NULL)
  ## create background
  barpos <- barplot(x, axes=FALSE, axisnames=FALSE, ylim=ylim,
      col="#ffffff", border=NA)
  ## draw a bar chart
  axis(1, at=barpos[,1], labels=names(x), Rcssclass="x")
  if (stripes) {
      stripevals <- axis(2, lwd=0, labels=NA)
      labpos <- axis(2, lwd=0, lwd.ticks=0, Rcssclass="y")
      abline(h=labpos)
  } else {
      axis(2, Rcssclass="y")
  }
  barplot(x, axes=FALSE, axisnames=FALSE, add=TRUE)
  mtext(main, Rcssclass="main")
  mtext(range.string(x), Rcssclass="submain")
  mtext(ylab, side=2, Rcssclass="ylab")
}
```

Two commands near the top fetch values for `stripe` and `ylim`. The subsequent code produces output conditional to these new variables (Figure 4A).

```r
rcss.barplot.4(a, main="Stripes", Rcss=style4)
```

The style we loaded also defines a class `dotted` (Figure 4B).

```r
rcss.barplot.4(a, main="Stripes, y-scale 100", Rcss=style4, Rcssclass="dotted")
```

In addition to providing styling for the horizontal rules, the class `dotted` also defines a property `ylim`. Its value is used within `rcss.barplot.5` to force limits on the vertical axis. This behavior can be desirable for several reasons. If the plotted values are proportions in percentages, it may be useful to show the full range from 0% to 100%. A fixed range can also be useful when displaying plots side-by-side (Figure 4C).

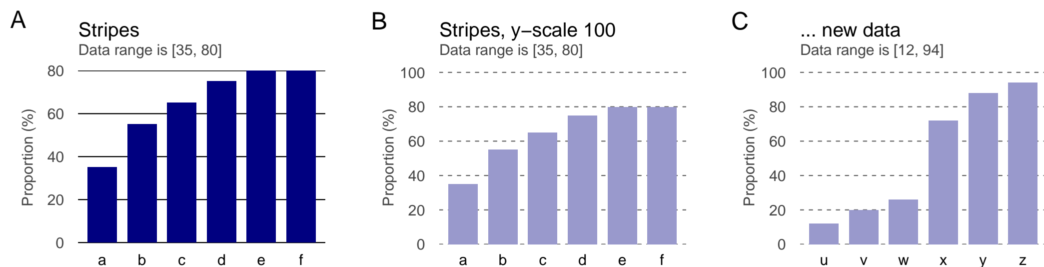


Figure 4: Charts using custom css selectors: (A) horizontal rules instead of a y-axis; (B) styled rules with a fixed vertical scale; (C) again styled rules with a fixed vertical scale, showing new data.

```
a2 <- setNames(c(12, 20, 26, 72, 88, 94), tail(letters, 6))
rcss.barplot.4(a2, main="... new data", Rcss=style4, Rcssclass="dotted")
```

The new data are easily compared with the old because the vertical scales in the charts are recognizably the same.

## 4.4 File watching

Developing a complex custom graphic requires much tinkering, i.e. defining settings, evaluating results, and adjusting the code as well as the style. This development cycle is simplified by the `RcssWatch` utility. This utility repeatedly evaluates a function, reloading code and style files before each iteration.

```
style.files = file.path("Rcss", paste0("vignettes.bar", c(1, 2, 3), ".Rcss"))
RcssWatch("rcss.barplot.4", files=style.files, x=a)
```

In this example, `RcssWatch` loads styles provided in `style.files` and evaluates `rcss.barplot.4` with argument `x=a`. It waits for a keystroke before repeating this procedure again. With this tool, adjustments in style definitions can be edited in Rcss files with a text editor, and the effects previewed with one keystroke. The utility also accepts files with extensions `.R` and `.r`, so it can help preview changes dues to the definition of the custom plot function.

# 5 Summary

This vignette introduced the Rcssplot package through an extended example based on a bar chart. We started with a visualization implemented using R's base graphics, and then adapted this design using Rcssplot.

At the technical level, the package provides a framework for customizing R graphics through a system akin to cascading style sheets. One part of the framework consists of functions that manage information in style sheets. These functions parse `Rcss` files, extract property/value pairs relevant in various contexts, and manage default styles and classes. Another part of the framework consists of functions that mimic base graphics functions (`plot`, `axis`, `text`, etc.), but extract styling details from the cascading style objects.

From a useability perspective, the Rcssplot package breaks building composite visualizations down into distinct tasks. Fine-tuning of aesthetics is delegated to cascading style sheets, which become external to R code. They can thus be adjusted safely without compromising data analysis and they can be shared between projects. The R code that is left is focused on data analysis and on the structure of the composite visualization. It is thus easier to understand and maintain.

The Rcssplot package is intended to provide a straightforward and familiar means to tune graphics (given background in conventional cascading-style sheets). It is important to note, however, that this is not the only graphics framework available for R. Indeed, other approaches have served as inspirations and models. In the space of static graphics, package `ggplot2` provides a mature approach to creating complex charts [1]. It supports tuning via themes; package `ggthemes` provides several examples [2]. In the space of interactive visualizations, packages `shiny` [3] and `plotly` [4] create very compelling results.

# Acknowledgements

Many thanks to R's documentation and manuals. A particularly valuable resource is [5].

Rcssplot is developed on github with contributions from (in alphabetical order): FrancoisGuillem, nfultz.


# References


[1] Hadley Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2009.

[2] Jeffrey B. Arnold. *ggthemes: Extra Themes, Scales and Geoms for 'ggplot2'*. R package version 3.3.0, 2016.

[3] Winston Chang and Joe Cheng and JJ Allaire and Yihui Xie and Jonathan McPherson. *shiny: Web Application Framework for R*. R package version 1.0.0, 2017.

[4] Carson Sievert and Chris Parmer and Toby Hocking and Scott Chamberlain and Karthik Ram and Marianne Corvellec and Pedro Despouy. *plotly: Create Interactive Web Graphics via 'plotly.js'* R package version 4.5.6, 2016.

[5] Hadley Wickham. *Advanced R*. http://adv-r.had.co.nz/


# A Appendix

## A.1 Compatibility with base graphics

All plot functions provided by Rcssplot are designed to mimic the familiar tools from base packages `graphics` and `grDevices`. In general, code that works under base graphics should continue to work even when Rcssplot is loaded. Thus, it should be straightforward to adopt the framework into an existing project. However, code that uses positional arguments can trigger errors. As an example, the following command does not work.

```
barplot(a, 2)
```

The cause for the error is the value '2', which is assumed to carry a meaning through its position in the function call. (In a barplot, the second argument is called `width` and can specify the width of individual bars).

There are two strategies to restore such code to working order. The first is to fall back on base graphics.

```
graphics::barplot(a, 2)
```

The second strategy is to add the intended argument names into the function call.

```
barplot(a, width=2)
```

Arguably, the second solution is clearer. Indeed, it is good practice to use argument names whenever calling complex functions.

## A.2 Grammar

Parsing of cascading style sheets is performed within the Rcssplot based on the grammar below.

```
stylesheet
   : [ ruleset ]*
   ;
 ruleset
   : simple_selector [ ',' simple_selector ]*
     '{' declaration? [ ';' declaration? ]* '}'
   ;
 simple_selector
   : IDENT [ class ]*
   | [ class ]+
```

```
  ;
class
  : '.' IDENT
  ;
declaration
  : property ':' expr
  ;
property
  : IDENT
  ;
expr
  : term [ term ]*
  ;
term
  : NUMBER | STRING | IDENT | HEXCOLOR
  ;
```

This formal definition is a summary and guide, and can serve as a comparison to the full css grammar of web design. However, actual parsing within the package is carried out manually, not using an auto-generated parser.

## A.3 Version history

### v1.0.0

- All Rcss wrappers change name to match functions from base graphics. They automatically mask base graphics functions.
- Watch utility.
- Miscellaneous convenience functions: `ctext`, `parplot`
- Error messages generated during parsing of Rcss files include line numbers in the original files.
- Some functions from 0.x versions are made redundant or deprecated. These provide warning messages, but will be removed in a future release.

### v0.3.0

- New functions for getting and setting values from Rcss objects: `RcssValue`, `RcssUpdate`. These functions are complementary to previously exisitng functions, but are less verbose, epsecially for fetching values from a default style.
- Better parsing and handling of special values in css files, e.g. TRUE/FALSE, NA, NULL.

### v0.2.0

- First version submitted to CRAN.

## A.4 Session info

```
sessionInfo()
```

```
## R version 3.5.1 (2018-07-02)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 16.04.6 LTS
##
## Matrix products: default
## BLAS: /software/opt/R/R-3.5.1/lib/libRblas.so
## LAPACK: /software/opt/R/R-3.5.1/lib/libRlapack.so
##
## locale:
##  [1] LC_CTYPE=en_GB.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_GB.UTF-8        LC_COLLATE=C
```

```
##  [5] LC_MONETARY=en_GB.UTF-8    LC_MESSAGES=en_GB.UTF-8
##  [7] LC_PAPER=en_GB.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_GB.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] Rcssplot_1.0.0 rmarkdown_1.13 knitr_1.23
##
## loaded via a namespace (and not attached):
##  [1] compiler_3.5.1   magrittr_1.5     htmltools_0.3.6  tools_3.5.1
##  [5] Rcpp_1.0.1       codetools_0.2-16 stringi_1.4.3    highr_0.8
##  [9] digest_0.6.20    stringr_1.4.0    xfun_0.7         evaluate_0.14
```