Styling R plots with cascading style sheets and Rcssplot

Tomasz Konopka

March 6, 2017

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 bookkeeping 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 a few 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 with made-up numbers.

```
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 our data, `a`, to obtain a chart with R's default visual style (Figure 1A).

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

The output contains 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 a little 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.

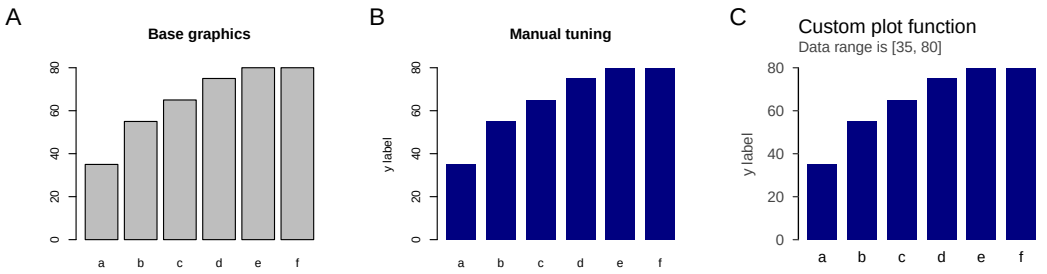

Figure 1: Charts created with base graphics using: (A) R's `barplot` function and default settings; (B) R's `barplot` function and some custom settings; (C) a custom plot function that styles bars, axes, and labels individually.

```
## 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 `x` 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. Although we can produce many charts by passing different data vectors and labels, 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 maintain 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.

3.1 Using Rcss styles

Let's adopt a convention where style definitions have `Rcss` extensions. Let's begin with a style file called `vignettes.bar1.Rcss`. This file is available in a sub-folder along with the package vignette.

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

The file contains 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 the style definition into the R environment using function `Rcss`.

```
library("Rcssplot")
style1 <- Rcss("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 newly loaded 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 wrappers for many of R's graphics functions. These wrappers have prefixes `Rcss` and accept the same arguments as their base-graphics parents. For example, to create a barplot, we invoke `Rcssbarplot` (Figure 2A).

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

When used in plain form as above, the output of the wrapper is exactly the same as from base graphics `barplot`. But we can add styling by passing our style object as an argument (Figure 2B).

```
Rcssbarplot(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 within a `barplot` function call (`border`, `col`, and `space`). The Rcssplot 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).

```
Rcssbarplot(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.

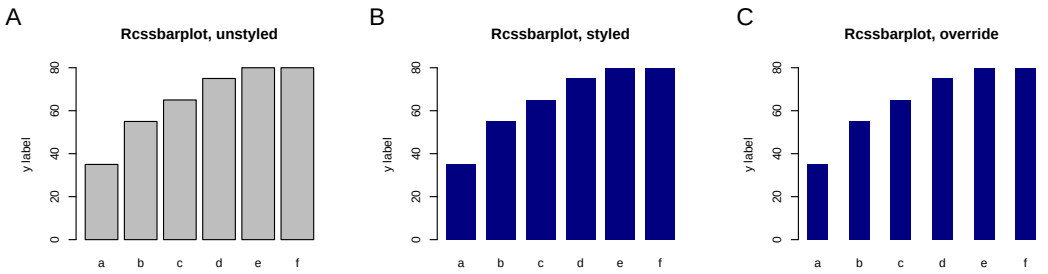

Figure 2: Charts created with base graphics and Rcssplot using: (A) the default style; (B) a style determined through a style sheet; (C) a style sheet, but with the bar width over-ridden by a setting within a function call.

3.2 Using 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. These are 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. It is worth noting a few features.

- 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. (As an aside, it is possible to format the style sheet in a more compact horizontal fashion following the usual conventions of css files).
- 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 `Rcssaxis`, 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 based on Rcssplot wrappers.

```
## 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 <- Rcssbarplot(x, axes=FALSE, axisnames=FALSE, Rcss=Rcss, Rcssclass=Rcssclass)
  ## add custom components
  Rcssaxis(1, at=barpos[,1], labels=names(x), Rcss=Rcss, Rcssclass=c(Rcssclass,"x"))
  Rcssaxis(2, Rcss=Rcss, Rcssclass=c(Rcssclass,"y"))
  Rcssmtext(main, Rcss=Rcss, Rcssclass=c(Rcssclass,"main"))
  Rcssmtext(range.string(x), Rcss=Rcss, Rcssclass=c(Rcssclass, "submain"))
  Rcssmtext(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. `Rcssaxis` 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(c("Rcss/vignettes.bar1.Rcss", "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, which we discuss next, is to define multiple styles within one `Rcss` object.

3.3 Using multiple styles

Let's look at another set `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 definition by providing the class name during the function call (Figure 3B).

```
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.

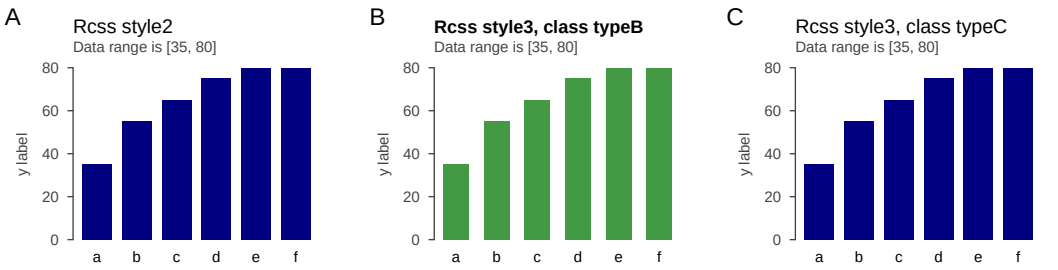

Figure 3: Charts created by custom plot functions with base graphics and Rcssplot using: (A) a style determined by css; (B) a style sub-class defined in css; (C) a style sub-class that is not defined in css (equivalent to (A)).

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 tricks that can simplify creation of custom graphics.

4 Additional features

This section covers some "advanced" features. The first three subsections deal with reducing repetitive code. The last subsection introduces usage of css objects as general data stores.

4.1 Overloading base graphics

Although our custom function `rcss.barplot.1` provides us with opportunities to tune the chart, its code has a number of inelegant repetitive elements. One of these is the `Rcss` prefix before each of the plot commands. It is possible to avoid this prefix by overloading the base graphics functions with their Rcssplot wrappers. Overloading is achieved using function `RcssOverload`.

```
## barplot using Rcssplot, version 2 (using overloading)
rcss.barplot.2 <- function(x, main="Custom Rcss plot", ylab="y label",
                           Rcss="default", Rcssclass=c()) {
  ## overload base graphics function by Rcssplot wrappers
  RcssOverload()
  ## create a barplot (without Rcss prefixes)
  barpos <- barplot(x, axes=FALSE, axisnames=FALSE, Rcss=Rcss, Rcssclass=Rcssclass)
  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"))
}
```

Here, the first step signals that subsequent calls to e.g. `axis` should actually invoke the corresponding wrappers, e.g. `Rcssaxis`. The subsequent code thus omits the `Rcss` prefixes.

Note that executing `RcssOverload` masks several commands from base graphics; the step carries numerous side effects for the working environment. Such behavior is typically undesirable. In this case, however, the net effect is similar as could be achieved by masking base graphics functions within the package and taking effect automatically when the package is loaded. This implementation with an explicit overload step provides a mechanism to activate the masking only when needed. It is provides a means to use both base graphics and Rcssplot wrappers within a single project.

4.2 Using a default style and compulsory classes

Other repetitive elements are the constructions `Rcss=Rcss` and `Rcssclass=Rcssclasss`. They ensure that the style object and the class specified through the function call are passed on to the individual wrappers. We can avoid this repetition by setting a default style and a compulsory class.

Handling of default values is achieved through objects `RcssDefaultStyle` and `RcssCompulsoryClass`. These objects can be defined in any environment, for example in the global environment in the console or inside a function. When present, the package wrappers detect them and use information therein to influence plot behavior. Consider, for example, the following custom function.

```
## barplot using Rcssplot, version 3 (using defaults & compulsory classes)
rcss.barplot.3 <- function(x, main="Custom Rcss plot", ylab="y label",
                           Rcss="default", Rcssclass=c()) {
  ## overload base graphics, set defaults and compulsory classes
  RcssOverload()
  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 here perform overloading, and then set a default style and compulsory class. Subsequent calls to graphics functions do not to refer to object `Rcss` or the class `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 to note that the preparation steps set `RcssDefaultStyle` and `RcssCompulsoryClass` with the help of function calls. Their use 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.

4.3 Using Rcssplot globally

In the previous two examples, `rcss.barplot.2` and `rcss.barplot.3`, we used overloading and changes to defaults within those custom functions, i.e. in environments local to those functions. In some cases, it may be reasonable to apply these changes in the global environment instead. This can be achieved by running the preparation outside of the custom function.

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

Subsequent to these commands, the custom function can be simplified further.

```
## barplot using Rcssplot, version 4 (assumes global use of Rcssplot)
rcss.barplot.4 <- 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")
}
```

There are a couple of points to note.

- Function `rcss.barplot.4` assumes that overloading has taken place. This is evidenced by calls to, for example, `axis`, with `Rcssclass` arguments. Thus, if this function is ever invoked without a prior overloading step, those calls will generate errors.
- 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 the argument is a mechanism that allows functions within a project to use different 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 is simply 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, `f002` 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. The use of the 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.

4.4 Using custom selectors

In this section, let's switch our focus toward using cascading style sheets as general data structures. From an abstract viewpoint, `Rcss` objects are just stores of 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 affect any of the Rcssplot wrapper functions. But we can write code to exploit information in `baraxis` by extracting values manually. The package provides two functions for this purpose, `RcssGetPropertyValue` and `RcssGetPropertyValueOrDefault`.

```
style4 <- Rcss(paste0("Rcss/vignettes.bar", c(1, 2, 4), ".Rcss"))
RcssGetPropertyValue(style4, "baraxis", "stripe")
```

```
## $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 provided default.

```
RcssGetPropertyValueOrDefault(style4, "baraxis", "stripe", default=0)
```

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

```
RcssGetPropertyValueOrDefault(style4, "baraxis", "strpe", default=0)
```

```
## [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 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.

```
## barplot using Rcssplot, version 5 (uses custom css selectors)
rcss.barplot.5 <- function(x, main="", ylab="Proportion (%)",
    Rcss="default", Rcssclass=c()) {
  ## use overloading, custom style, compulsary class
  RcssOverload()
  RcssDefaultStyle <- RcssGetDefaultStyle(Rcss)
  RcssCompulsoryClass <- RcssGetCompulsoryClass(Rcssclass)
  ## extract custom properties - show axis? force ylim?
```

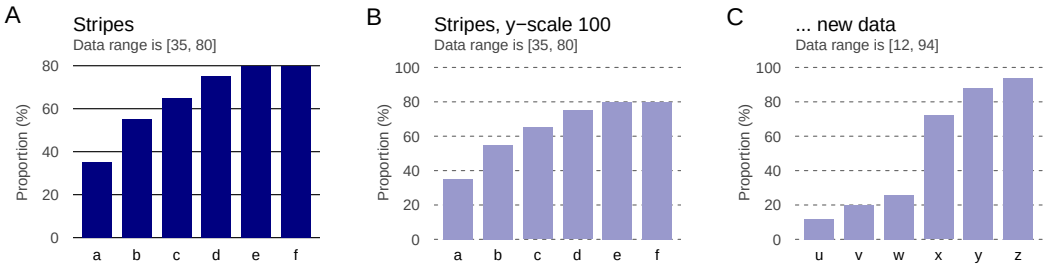

Figure 4: Charts using custom css selectors: (A) horizontal rules instead of a y-axis; (B) styled rules with a fixed vertical scale; (C) again styled rules with a fixed vertical scale, but with a different data input.

```
  stripes <- RcssGetPropertyValueOrDefault(Rcss, selector="baraxis",
    property="stripe", default=0)
  ylim <- RcssGetPropertyValueOrDefault(Rcss, selector="baraxis",
    property="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 `stripes` and `ylim`. The subsequent code produces output conditional to these new variables (Figure 4A).

```
rcss.barplot.5(a, main="Stripes", Rcss=style4)
```

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

```
rcss.barplot.5(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).

```
a2 <- setNames(c(12, 20, 26, 72, 88, 94), tail(letters, 6))
rcss.barplot.5(a2, main="... new data", Rcss=style4, Rcssclass="dotted")
```

In this example, the new data are easily compared with the old because the vertical scales in the charts are recognizably the same.

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 wrapper functions that mimick 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): cuche27, 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 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.2 Session info

```
sessionInfo()
```

```
## R version 3.3.1 (2016-06-21)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 14.04.5 LTS
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=C
##  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] knitr_1.14       Rcssplot_0.2.0.0
##
## loaded via a namespace (and not attached):
## [1] magrittr_1.5  formatR_1.4   tools_3.3.1   stringi_1.1.1 highr_0.6
## [6] stringr_1.1.0 evaluate_0.9
```