YaRrr!

The Pirate’s Guide to R
YARRR! THE PIRATE’S GUIDE TO R
7: Importing, saving, and managing data 103
   The working directory 104
   The workspace 105
   Saving and loading data with .RData files 107
   Saving and loading data as .txt files 109
   Additional Tips 112
   Test your R Might! 113

8: Advanced dataframe manipulation 115
   Merging dataframes with merge() 117
   aggregate() 119
   dplyr 122
   Additional Tips 124
   Test your R Might!: Mmmmm...caffeine 125

9: Plotting: Part 1 127
   How does R manage plots? 127
   Color basics 129
   Scatterplot: plot() 132
   Histogram: hist() 134
   Barplot: barplot() 135
   The Pirate Plot: pirateplot() 138
   Low-level plotting functions 142
   Adding new points to a plot with points() 143
   Adding straight lines with abline() 144
   Adding text to a plot with text() 146
   Combining text and numbers with paste() 147
   Additional low-level plotting functions 151
   Saving plots to a file 152
   Test your R Might! Purdy pictures 154
10: Plotting: Part Deux 157
  Advanced colors 157
  Plot margins 162
  Arranging multiple plots with par(mfrow) and layout 163
  Additional Tips 166

11: Inferential Statistics: 1 and 2-sample Null-Hypothesis tests 167
  Null vs. Alternative Hypotheses, Descriptive Statistics, Test Statistics, and p-values: A very short introduction 168
  Null v Alternative Hypothesis 168
  Hypothesis test objects – htest 172
  T-test with t.test() 174
  Correlation test with cor.test() 178
  Chi-square test 181
  Getting APA-style conclusions with the apa function 183
  Test your R might! 185

12: ANOVA and Factorial Designs 187
  Between-Subjects ANOVA 188
  4 Steps to conduct a standard ANOVA in R 190
  ANOVA with interactions: (y  \sim  x1 * x2) 194
  Additional tips 197
  Test your R Might! 200

13: Regression 201
  The Linear Model 201
  Linear regression with lm() 201
  Estimating the value of diamonds with lm() 202
  Including interactions in models: dv  \sim  x1 * x2 206
  Comparing regression models with anova() 208
  Regression on non-Normal data with glm() 211
9: Plotting: Part 1

Sammy Davis Jr. was one of the greatest American performers of all time. If you don’t know him already, Sammy was an American entertainer who lived from 1925 to 1990. The range of his talents was just incredible. He could sing, dance, act, and play multiple instruments with ease. So how is R like Sammy Davis Jr? Like Sammy Davis Jr., R is incredibly good at doing many different things. R does data analysis like Sammy dances, and creates plot like Sammy sings. If Sammy and R did just one of these things, they’d be great. The fact that they can do both is pretty amazing.

How does R manage plots?

When you evaluate plotting functions in R, R can build the plot in different locations. The default location for plots is in a temporary plotting window within your R programming environment. In RStudio, plots will show up in the Plot window (typically on the bottom right hand window pane). In Base R, plots will show up in a Quartz window.

You can think of these plotting locations as canvases. You only have one canvas active at any given time, and any plotting command you run will put more plotting elements on your active canvas. Certain high–level plotting functions like plot() and hist() create brand new canvases, while other low–level plotting functions like points() and segments() place elements on top of existing canvases.

Don’t worry if that’s confusing for now – we’ll go over all the details soon.

Let’s start by looking at a basic scatterplot in R using the plot() function. When you execute the following code, you should see a plot open in a new window:

```r
# A basic scatterplot

plot(x = 1:10,
     y = 1:10,
     types = "p",
     main = "Basic Scatterplot",
     xlab = "X-axis",
     ylab = "Y-axis")
```
Let’s take a look at the result. We see an x–axis, a y–axis, 10 data points, an x–axis label, a y–axis label, and a main plot title. Some of these items, like the labels and data points, were entered as arguments to the function. For example, the main arguments x and y are vectors indicating the x and y coordinates of the (in this case, 10) data points. The arguments xlab, ylab, and main set the labels to the plot. However, there were many elements that I did not specify – from the x and y axis limits, to the color of the plotting points. As you’ll discover later, you can change all of these elements (and many, many more) by specifying additional arguments to the plot() function. However, because I did not specify them, R used default values – values that R uses unless you tell it to use something else.

For the rest of this chapter, we’ll go over the main plotting functions, along with the most common arguments you can use to customize the look of your plot.
Color basics

Most plotting functions have a color argument (usually col) that allows you to specify the color of whatever your plotting. There are many ways to specify colors in R, let’s start with the easiest ways.

Specifying simple colors

col = "red", col = "blue", col = "lawngreen" (etc.)
The easiest way to specify a color is to enter its name as a string. For example col = "red" is R’s default version of the color red. Of course, all the basic colors are there, but R also has tons of quirky colors like snow, papayawhip and lawngreen. To see all color names in R, run the code

```r
colors() # Show me all the color names!
```

col = gray(level, alpha)
The gray() function takes two arguments, level and alpha, and returns a shade of gray. For example, gray(level = 1) will return white. The alpha argument specifies how transparent to make the color on a scale from 0 (completely transparent), to 1 (not transparent at all). The default value for alpha is 1 (not transparent at all.)

Transparent Colors with transparent()

I don’t know about you, but I almost always find transparent colors to be more appealing than solid colors. Not only do they help you see when multiple points are overlapping, but they’re just much nicer to look at. Just look at the overlapping circles in the plot below.
Unfortunately, as far as I know, base-R does not make it easy to make transparent colors. Thankfully, there is a function in the \texttt{yarrr} package called \texttt{transparent} that makes it very easy to make any color transparent. To use it, just enter the original color as the main argument (\texttt{orig.col}), then enter how transparent you want to make it (from 0 to 1) as the second argument (\texttt{trans.val}). You can either save the transparent color as a new color object, or use the function directly in a plotting function like I do in the scatterplot in the margin.

```r
library(yarrr)  # load the yarrr package

# Make red.t.5, a transparent version of red
red.t.5 <- transparent(orig.col = "red",
                        trans.val = .5)

# Make blue.t.5, a transparent version of blue
blue.t.5 <- transparent(orig.col = "blue",
                        trans.val = .5)

# Plot with Standard Colors
plot(x = pirates$height, y = pirates$weight, col = "blue", pch = 16, main = "Standard Colors")

# Plot with Transparent Colors
library(yarrr)  # Load the yarrr package

# To get the transparent() function

plot(x = pirates$height, y = pirates$weight, col = transparent("blue", trans.val = .9), pch = 16, main = "Transparent Colors")
```

Later on in the book, we’ll cover more advanced ways to come up with colors using color palettes (using the \texttt{RColorBrewer} package or the \texttt{piratepal()} function in the \texttt{yarrr} package) and functions that generate shades of colors based on numeric data (like the \texttt{colorRamp2()} function in the \texttt{circlize} package).
**High vs. low-level plotting commands**

There are two general types of plotting commands in R: high and low-level. High level plotting commands, like `plot()`, `hist()` and `pirateplot()` create entirely new plots. Within these high level plotting commands, you can define the general layout of the plot - like the axis limits and plot margins.

Low level plotting commands, like `points()`, `segments()`, and `text()` add elements to existing plots. These low level commands don’t change the overall layout of a plot - they just add to what you’ve already created. Once you are done creating a plot, you can export the plot to a pdf or jpeg using the `pdf()` or `jpeg()` functions. Or, if you’re creating documents in Markdown or Latex, you can add your plot directly to your document.

**Plotting arguments**

Most plotting functions have tons of optional arguments (also called parameters) that you can use to customize virtually everything in a plot. To see all of them, look at the help menu for `par` by executing `?par`. However, the good news is that you don’t need to specify all possible parameters you create a plot. Instead, there are only a few critical arguments that you must specify - usually one or two vectors of data. For any optional arguments that you do not specify, R will use either a default value, or choose a value that makes sense based on the data you specify.

In the following examples, I will cover the main plotting parameters for each plotting type. However, the best way to learn what you can, and can’t, do with plots, is to try to create them yourself!

I think the best way to learn how to create plots is to see some examples. Let’s start with the main high-level plotting functions.
**Scatterplot: plot()**

The most common high-level plotting function is `plot(x, y)`. The `plot()` function makes a scatterplot from two vectors `x` and `y`, where the `x` vector indicates the x (horizontal) values of the points, and the `y` vector indicates the y (vertical) values.

```r
plot(x = 1:10, # x-coordinates
     y = 1:10, # y-coordinates
     type = "p", # Draw points (not lines)
     main = "My First Plot",
     xlab = "This is the x-axis label",
     ylab = "This is the y-axis label",
     xlim = c(0, 11), # Min and max values for x-axis
     ylim = c(0, 11), # Min and max values for y-axis
     col = "blue", # Color of the points
     pch = 16, # Type of symbol (16 means Filled circle)
     cex = 1 # Size of the symbols
)
```

Here are some of the main arguments to `plot()`:

- `x, y` Vectors specifying the x and y values of the points
- `type` Type of plot. "l" means lines, "p" means points, "b" means lines and points, "n" means no plotting
- `main` Label for the plot title
- `xlab` Label for the x-axis
- `ylab` Labels for the y-axis
- `xlim` Limits to the x-axis. For example, `xlim = c(0, 100)` will set the minimum and maximum of the x-axis to 0 and 100.
- `ylim` Limits to the y-axis. For example, `ylim = c(50, 200)` will set the minimum and maximum of the y-axis to 50 and 200.
- `pch` An integer indicating the type of plotting symbols (see `?points` and section below), or a string specifying symbols as text. For example, `pch = 21` will create a two-color circle, while `pch = "p"` will plot the character “P”. To see all the different symbol types, run `?points`.
- `col` Main color of the plotting symbols. For example `col = "red"` will create red symbols.
- `bg` Color of the background of two-color symbols 21 through 25. For example `pch = 21, bg = "blue"` will set the background of the two-color circle to Blue.
- `cex` A numeric vector specifying the size of the symbols (from 0 to Inf). The default size is 1. `cex = 2` will make the points very large, while `cex = .5` will make them very small.

Aside from the `x` and `y` arguments, all of the arguments are optional. If you don’t specify a specific argument, then R will use a
default value, or try to come up with a value that makes sense. For example, if you don’t specify the xlim and ylim arguments, R will set the limits so that all the points fit inside the plot.

**Symbol types: pch**

When you create a plot with `plot()` (or points with `points()`), you can specify the type of symbol with the pch argument. You can specify the symbol type in one of two ways: with an integer, or with a string. If you use a string (like "p"), R will use that text as the plotting symbol. If you use an integer value, you’ll get the symbol that correspond to that number. See Figure 45 for all the symbol types you can specify with an integer.

Symbols differ in their shape and how they are colored. Symbols 1 through 14 only have borders and are always empty, while symbols 15 through 20 don’t have a border and are always filled. Symbols 21 through 25 have both a border and a filling. To specify the border color or background for symbols 1 through 20, use the col argument. For symbols 21 through 25, you set the color of the border with col, and the color of the background using bg.

Let’s look at some different symbol types in action:

\[
\begin{align*}
pch = 2, & \quad col = 'blue' \\
pch = 16, & \quad col = 'orchid2' \\
pch = 21, & \quad col = 'black', \quad bg = 'orangered2' \\
pch = 25, & \quad col = 'pink3', \quad bg = 'plum3'
\end{align*}
\]

Figure 45: The symbol types associated with the pch plotting parameter.
**Histogram: hist()**

Histograms are the most common way to plot unidimensional numeric data. To create a histogram we’ll use the `hist()` function. The main argument to `hist()` is `x`, a vector of numeric data. If you want to specify how the histogram bins are created, you can use the `breaks` argument. To change the color of the border or background of the bins, use `col` and `border`:

Let’s create a histogram of the weights in the ChickWeight dataset:

```r
hist(x = ChickWeight$weight, 
    main = "Chicken Weights", 
    xlab = "Weight", 
    xlim = c(0, 500))
```

We can get more fancy by adding additional arguments like `breaks = 20` to force there to be 20 bins, and `col = "papayawhip"` and `bg = "hotpink"` to make it a bit more colorful (see the margin figure 46)

If you want to plot two histograms on the same plot, for example, to show the distributions of two different groups, you can use the `add = T` argument to the second plot. See Figure 47 to see this in action.

```r
hist(x = ChickWeight$weight[ChickWeight$Diet == 1], 
    main = "Two Histograms in one", 
    xlab = "Weight", 
    ylab = "Frequency", 
    breaks = 20, 
    xlim = c(0, 500), 
    col = gray(0, .5))

hist(x = ChickWeight$weight[ChickWeight$Diet == 2], 
    breaks = 30, 
    add = TRUE, # Add plot to previous one! 
    col = gray(1, .5))
```
*Barplot: barplot()*

A barplot is good for showing summary statistics for different groups. The primary argument to a barplot is `height`: a vector of numeric values which will generate the height of each bar. To add names below the bars, use the `names.arg` argument. For additional arguments specific to `barplot()`, look at the help menu with `?barplot`:

```r
barplot(height = 1:5, names.arg = c("G1", "G2", "G3", "G4", "G5"), main = "Example Barplot", xlab = "Group", ylab = "Height")
```

Of course, you should plot more interesting data than just a vector of integers with a barplot. In the margin figure, I create a barplot with the average weight of chickens for each time point with yellow bars. If you want to plot different colored bars from different datasets, create one normal barplot, then create a second barplot with the `add = T` argument. In Figure 48, I plotted the average weights for chickens on Diets 1 and 2 separately on the same plot.
Clustered barplot

If you want to create a clustered barplot, with different bars for different groups of data, you can enter a matrix as the argument to height. R will then plot each column of the matrix as a separate set of bars. For example, let’s say I conducted an experiment where I compared how fast pirates can swim under four conditions: Wearing clothes versus being naked, and while being chased by a shark versus not being chased by a shark. Let’s say I conducted this experiment and calculated the following average swimming speed:

<table>
<thead>
<tr>
<th></th>
<th>Naked</th>
<th>Clothed</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Shark</td>
<td>2.1</td>
<td>1.5</td>
</tr>
<tr>
<td>Shark</td>
<td>3.0</td>
<td>3.0</td>
</tr>
</tbody>
</table>

Table 1: Mean Swimming times (in meters / second)

I can represent these data in a matrix as follows. In order for the final barplot to include the condition names, I’ll add row and column names to the matrix with colnames() and rownames():

swim.data <- cbind(c(2.1, 3), c(1.5, 3)) # Naked Times
# Clothed Times

colnames(swim.data) <- c("Naked", "Clothed")
rownames(swim.data) <- c("No Shark", "Shark")

Here’s how the final matrix looks:

```
swim.data

##                  Naked  Clothed
## No Shark          2.1  1.5
## Shark             3.0  3.0
```

Now, when I enter this matrix as the height argument to barplot(), I’ll get multiple bars.

```r
barplot(height = swim.data,
        beside = T,  # Put the bars next to each other
        legend.text = T,  # Add a legend
        col = c(transparent("green", .7),
                 transparent("red", .7)),
        main = "Swimming Speed Experiment",
        ylab = "Speed (in meters / second)",
        xlab = "Clothing Condition",
        ylim = c(0, 3.5))
```
Swimming Speed Experiment

<table>
<thead>
<tr>
<th>Clothing Condition</th>
<th>Speed (in meters/second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naked</td>
<td>3.0</td>
</tr>
<tr>
<td>Clothed</td>
<td>0.5</td>
</tr>
</tbody>
</table>
The Pirate Plot: pirateplot()

A pirateplot is a new type of plot written just for this book! The pirateplot is an easy-to-use function that, unlike barplots and boxplots, can easily show raw data, descriptive statistics, and inferential statistics in one plot. Here are the four key elements in a pirateplot:

![4 Elements of a pirateplot](image)

- Points Raw Data
- Bar/Line Center
- Bean Density
- Band Inference 95% HDI or CI

The two main arguments to `pirateplot()` are `formula` and `data`. In `formula`, you specify plotting variables in the form `dv ~ iv`, where `dv` is the name of the dependent variable, and `iv` is the name of the independent variable. In `data`, you specify the name of the dataframe object where the variables are stored.

The `pirateplot()` function is part of the `yarrr` package. So to use it, you’ll first need to install the `yarrr` package.
Let’s create a pirateplot of the ChickWeight data. I’ll set the dependent variable to weight, and the independent variable to Diet.

```r
library("yarrr")
pirateplot(formula = weight ~ Diet, # dv is weight, iv is Diet
data = ChickWeight,
   main = "pirateplot of chicken weights",
   xlab = "Diet",
   ylab = "Weight")
```

As you can see, the pirateplot shows us the complete distribution of data for each diet. In addition, because we have inference bands showing 95% Highest Density Intervals (HDIs), we can make inferential comparisons between groups. For example, because the intervals between Diets 1 and 3 do not overlap, we can confidently conclude that Diet 3 lead to credibly higher weights compared to Diet 1.

Here are some of the main arguments to `pirateplot()`:

- **formula**: A formula in the form `dv ~ iv1 + iv2` where `dv` is the name of the dependent variable, and `iv1` and `iv2` are the name(s) of the independent variable(s). Up to two independent variables can be used.
- **data**: A dataframe containing the dependent and independent variables.
- **pal**: A string indicating the color palette of the plot. Can be a single color, or the name of a palette in the `piratepal()` function (e.g., "basel", "google", "xmen")
- **theme**: An integer indicating what plotting theme to use. As of yarrr version 0.1.2, versions 0, 1, 2, and 3 are supported.
The `pirateplot` function contains a few different plotting themes. To change the theme, use the `theme` argument. Here is a `pirateplot` using `theme = 2`:

```r
pirateplot(formula = weight ~ Diet, # dv is weight, iv is Diet
data = ChickWeight,
theme = 2,
main = "pirateplot theme 2",
lab = "Time",
lab = "Weight",
gl.col = "gray") # Add gray gridlines
```

You can easily change the colors in a `pirateplot` with the `pal` argument. Setting `pal = "black"` will create a black and white `pirateplot`. Or, setting `pal = "pony"` will use the My Little Pony theme. See the plots on the right margin for examples.

You can include up to three independent variables in the formula argument to the `piratepal()` function to create a souped up clustered barplot. For example, the `ToothGrowth` dataframe measures the length of Guinea Pig’s teeth based on different supplements and different doses. We can plot both as follows:
# Plotting data from two IVs in pirateplot

```r
pirateplot(formula = len ~ dose + supp, # Two IVs
data = ToothGrowth,
theme = 2,
gl.col = "gray",
xlab = "Dose",
ylab = "Tooth Length",
main = "ToothGrowth Data")
```

There are many additional optional arguments to `pirateplot()` – for example, you can use different color palettes with the `pal` argument or change how transparent different elements are with `inf.f.o`, `bean.f.o` and others. To see them all, look at the help menu with `?pirateplot`. 
Low-level plotting functions

Once you’ve created a plot with a high-level plotting function, you can add additional elements with low-level functions. You can add data points with `points()`, reference lines with `abline()`, text with `text()`, and legends with `legend()`.

Starting with a blank plot

Before you start adding elements with low-level plotting functions, it’s useful to start with a blank plotting space. To do this, execute the `plot()` function, but use the `type = "n"` argument to tell R that you don’t want to plot anything yet. Once you’ve created a blank plot, you can additional elements with low-level plotting commands.

```r
plot(x = 1,
     xlab = "X Label", ylab = "Y Label",
     xlim = c(0, 100), ylim = c(0, 100),
     main = "Blank Plotting Canvas",
     type = "n")
```

![Blank Plotting Canvas](image)

Figure 50: Ahhhh...a fresh blank canvas waiting for low-level plotting functions.
Adding new points to a plot with `points()`

To add new points to an existing plot, use the `points()` function.25

Let’s use `points()` to create a plot with different symbol types for different data. I’ll use the pirates dataset and plot the relationship between a pirate’s age and the number of tattoos he/she has. I’ll create separate points for male and female pirates:

```r
library(yarr) # Load the yarr package (for pirates dataset)

# Create the plot with male data
plot(x = pirates$age[pirates$sex == "male"],
     y = pirates$tattoos[pirates$sex == "male"],
     xlim = c(10, 45), ylim = c(0, 22),
     pch = 16,
     col = transparent("coral2", trans.val = .2),
     xlab = "Age", ylab = "Number of Tattoos",
     main = "Pirate Tattoos by Age\nMale vs. Female"
)
```

Now, I’ll add points for the female data with `points`:

```r
# Add points for female data
points(x = pirates$age[pirates$sex == "female"],
       y = pirates$tattoos[pirates$sex == "female"],
       pch = 16,
       col = transparent("steelblue3", trans.val = .3)
)
```

My first plotting command with `plot()` will create the left-hand figure below. The second plotting command with `points()` will add to the plot and create the right-hand figure below.

---

25 The `points()` function has many similar arguments to the `plot()` function, like `x` (for the x-coordinates), `y` (for the y-coordinates), and parameters like `col` (border color), `cex` (point size), and `pch` (symbol type). To see all of them, look at the help menu with `?points()`.
Adding straight lines with abline()

To add straight lines to a plot, use abline() or segments(). abline() will add a line across the entire plot, while segments() will add a line with defined starting and end points.

For example, we can add reference lines to a plot with abline(). In the following plot, I’ll add vertical and horizontal reference lines showing the means of the variables on the x and y axes:

```r
plot(x = pirates$weight, 
     y = pirates$height, 
     main = "Adding reference lines with abline", 
     pch = 16, col = gray(0, .1))

# Add horizontal line at mean height
abline(h = mean(pirates$height))

# Add vertical line at mean weight
abline(v = mean(pirates$weight))
```

You can use abline() to add gridlines to a plot. To do this, enter the locations of horizontal lines with the h argument, and vertical lines with the v argument (See Figure 51):

To change the look of your lines, use the lty argument, which
changes the type of line (see Figure 53), `lwd`, which changes its thickness, and `col` which changes its color

`segments()`

The `segments()` function works very similarly to `abline()` – however, with the `segments()` function, you specify the beginning and end points of the segments. Here, I’ll use `segments()` to connect two vectors of data:

```r
# Before and after data
before <- c(2.1, 3.5, 1.8, 4.2, 2.4, 3.9, 2.1, 4.4)
after <- c(7.5, 5.1, 6.9, 3.6, 7.5, 5.2, 6.1, 7.3)

# Create plotting space and before scores
plot(x = rep(1, length(before)),
     y = before,
     xlim = c(.5, 2.5),
     ylim = c(0, 11),
     ylab = "Score",
     xlab = "Time",
     main = "Using segments() to connect points",
     xaxt = "n")

# Add after scores
points(x = rep(2, length(after)), y = after)

# Now add connections with segments()
segments(x0 = rep(1, length(before)),
         y0 = before,
         x1 = rep(2, length(after)),
         y1 = after,
         col = gray(0, .5))

# Add labels
mtext(text = c("Before", "After"),
       side = 1, at = c(1, 2), line = 1)
```

You can see the resulting figure in the margin.
Adding text to a plot with text()

With text(), you can add text to a plot. You can use text() to highlight specific points of interest in the plot, or to add information (like a third variable) for every point in a plot. I’ve highlighted some of the key arguments to text() in Figure 54.

For example, the following code adds the three words "Put", "Text", and "Here" at the coordinates (1, 9), (5, 5), and (9, 1) respectively. See Figure 55 for the plot:

```r
plot(1, xlim = c(0, 10), ylim = c(0, 10), type = "n")

text(x = c(1, 5, 9),
     y = c(9, 5, 1),
     labels = c("Put", "text", "here"))
```

You can do some cool things with text(). I’ll create a scatterplot of data, and add data labels above each point:

```r
height <- c(156, 175, 160, 172, 159)
weight <- c(65, 74, 69, 72, 66)
id <- c("p1", "p2", "p3", "p4", "p5")

plot(x = height, y = weight,
     xlim = c(155, 180), ylim = c(65, 80))

text(x = height, y = weight,
     labels = id, pos = 3)
```

Main arguments to text():
- x, y: The location(s) of the text
- labels: The text to plot
- cex: The size of the text
- adj: How should the text be justified? 0 = left justified, 0.5 = centered, 1 = right justified
- pos: The position of the text relative to the x and y coordinates. Values of 1, 2, 3, and 4 put the text below, to the left, above, and to the right of the x-y coordinates respectively.

Figure 54: Main arguments to the text() function

```r
plot(1, xlim = c(0, 10), ylim = c(0, 10), type = "n")
text(x = c(1, 5, 9),
     y = c(9, 5, 1),
     labels = c("Put", "text", "here"))
```

Figure 55: Adding text to a plot. The characters in the argument to labels will be plotted at the coordinates indicated by x and y.
When entering text in the labels argument, keep in mind that R will, by default, plot the entire text in one line. However, if you are adding a long text string (like a sentence), you may want to separate the text into separate lines. To do this, add the text "\n" where you want new lines to start. Look at Figure 56 for an example.

**Combining text and numbers with `paste()`**

A common way to use text in a plot, either in the main title of a plot or using the `text()` function, is to combine text with numerical data. For example, you may want to include the text "Mean = 3.14" in a plot to show that the mean of the data is 3.14. But how can we combine numerical data with text? In R, we can do this with the `paste()` function:

The paste function will be helpful to you anytime you want to combine either multiple strings, or text and strings together. For example, let’s say you want to write text in a plot that says *The mean of these data are XXX*, where XXX is replaced by the group mean. To do this, just include the main text and the object referring to the numerical mean as arguments to `paste()`:

```r
data <- ChickWeight$weight
mean(data)
## [1] 122
paste("The mean of data is", mean(data)) # No rounding
## [1] "The mean of data is 121.81339100346"
paste("The mean of data is", round(mean(data), 2)) # No rounding
## [1] "The mean of data is 121.82"
```

Let’s create a plot with labels using the `paste()` function. We’ll plot the chicken weights over time, and add text to the plot specifying the overall mean and standard deviations of weights.

```r
# Create the plot
plot(x = ChickWeight$Time,
     y = ChickWeight$weight,
     col = gray(.3, .5),
     pch = 16,
     main = "Chicken Weights")

# Add text
```

To plot text on separate lines in a plot, put the tag "\n" between lines.

```r
plot(..., main = "The |n tag",
     xlab = "", ylab = "")
```

The 
 tag
Text without 
Haikus are easy. But sometimes they don't make sense. Refrigerator
Text with 
Haikus are easy
But sometimes they don't make sense
Refrigerator

Figure 56: Using the "\n" tag to plot text on separate lines.

When you include descriptive statistics in a plot, you will almost always want to use the `round(x, digits)` function to reduce the number of digits in the statistic.
text(x = 0, y = 300, labels = paste("Mean weight = ", round(mean(ChickWeight$weight), 2), 
        "\nStandard deviation = ", round(sd(ChickWeight$weight), 2), 
        sep = ""), adj = 0)
The `curve()` function allows you to add a line showing a specific function or equation to a plot.

### Parameters

- **expr**
  - The name of a function written as a function of x that returns a single vector. You can either use base functions in R like `expr = x^2`, `expr = x + 4 - 2`, or use your own custom functions such as `expr = my.fun`, where `my.fun` is previously defined (e.g.; `my.fun <- function(x) dnorm(x, mean = 10, sd = 3)`)

- **from, to**
  - The starting (from) and ending (to) value of x to be plotted.

- **add**
  - A logical value indicating whether or not to add the curve to an existing plot. If `add = FALSE`, then `curve()` will act like a high-level plotting function and create a new plot. If `add = TRUE`, then `curve()` will act like a low-level plotting function.

- **lty, lwd, col**
  - Additional arguments such as `lty`, `col`, `lwd`, ...

For example, to add the function `x^2` to a plot from the x-values -10 to 10, you can run the code:

```r
curve(expr = x^2, from = -10, to = 10)
```

If you want to add a custom function to a plot, you can define the function and then use that function name as the argument to `expr`. For example, to plot the normal distribution with a mean of 10 and standard deviation of 3, you can use this code:

```r
my.fun <- function(x) {dnorm(x, mean = 10, sd = 3)}
curve(expr = my.fun, from = -10, to = 10)
```

In Figure 57, I use the `curve()` function to create curves of several mathematical formulas.
The last low-level plotting function that we’ll go over in detail is `legend()` which adds a legend to a plot. This function has the following arguments

- **x, y**
  Coordinates of the legend - for example, `x = 0, y = 0` will put the text at the coordinates `(0, 0)`. Alternatively, you can enter a string indicating where to put the legend (i.e.; "topright", "topleft"). For example, "bottomright" will always put the legend at the bottom right corner of the plot.

- **labels**
  A string vector specifying the text in the legend. For example, `legend = c("Males", "Females")` will create two groups with names Males and Females.

- **pch, lty, lwd, col, pt.bg, ...**
  Additional arguments specifying symbol types (`pch`), line types (`lty`), line widths (`lwd`), background color of symbol types 21 through 25 (`pt.bg`) and several other optional arguments. See ?legend for a complete list.

For example, to add a legend to bottom-right of an existing graph where data from females are plotted in blue circles and data from males are plotted in pink circles, you’d use the following code:

```r
legend("bottomright", # Put legend in bottom right of graph
  legend = c("Females", "Males"), # Names of groups
  col = c("blue", "orange"), # Colors of symbols
  pch = c(16, 16) # Point types
)
```

In margin Figure 58 I use this code to add a legend to plot containing data from males and females.
Additional low-level plotting functions

There are many more low-level plotting functions that can add additional elements to your plots. Here are some I use. To see examples of how to use each one, check out their associated help menus.

```
par(mar=c(5, 4, 4, 2))
plot(1, xlim=c(0, 100), ylim=c(0, 100),
type="n", xaxt="n", yaxt="n",
ylab="", xlab="", main="Adding simple figures to a plot")
text(25, 95, labels="rect()")
rect(xleft = 10, ybottom = 78,
     xright = 40, ytop = 90, lwd = 2, col = "coral")
text(25, 60, labels="polygon()")
polygon(x = runif(0, 15, 30),
y = runif(0, 40, 90),
col = "skyblue")
# polygon(x = c(15, 15, 25, 15),
#        y = c(40, 40, 55, 40),
#        col = "skyblue")
# }
text(25, 30, labels = "segments()")
segments(x0 = runif(1, 10, 40),
y0 = runif(1, 5, 25),
x1 = runif(1, 10, 40),
y1 = runif(1, 5, 25), lwd = 2)
text(75, 95, labels = "symbols(circles)"
symbols(x = runif(20, 10, 90),
y = runif(20, 10, 70),
circles = c(0.1, 1, 3),
add = T, bg = gray(0.5), lty = 1)
text(75, 30, labels = "arrows()")
arrows(x0 = runif(10, 10, 80),
y0 = runif(10, 10, 25),
x1 = runif(10, 80, 90),
y1 = runif(10, 15, 25),
length = 1, lwd = 2)
```

**rect()**

Add rectangles to a plot at coordinates specified by xleft, ybottom, xright, ybottom. For example, to add a rectangle with corners at (0, 0) and (10, 10), specify xleft = 0, ybottom = 0, xright = 10, ytop = 10. Additional arguments like col, border change the color of the rectangle.

**polygon()**

Add a polygon to a plot at coordinates specified by vectors x and y. Additional arguments such as col, border change the color of the inside and border of the polygon.

**segments(), arrows()**

Add segments (lines with fixed endings), or arrows to a plot.

**symbols(add = T)**

Add symbols (circles, squares, rectangles, stars, thermometers) to a plot. The dimensions of each symbol are specified with specific input types. See `symbols` for details. Specify add = T to add to an existing plot or add = F to create a new plot.

**axis()**

Add an additional axis to a plot (or add fully customizable x and y axes). Usually you only use this if you set xaxt = "n", yaxt = "n" in the original high-level plotting function.

**mtext()**

Add text to the margins of a plot. Look at the help menu for `mtext()` to see parameters for this function.
Saving plots to a file

Once you’ve created a plot in R, you may wish to save it to a file so you can use it in another document. To do this, you’ll use either the \texttt{pdf()} or \texttt{jpeg()} functions. These functions will save your plot to either a .pdf or jpeg file.

\begin{quote}
\begin{center}
\textbf{pdf()} and \texttt{jpeg()}
\end{center}
\end{quote}

\texttt{pdf()}
\begin{quote}
The name and file destination of the final plot entered as a string. For example, to put a plot on my desktop, I’d write \texttt{file = \\
"/Users/nphillips/Desktop/plot.pdf"} when creating a pdf, and \texttt{file = \\
"/Users/nphillips/Desktop/plot.jpg"} when creating a jpeg.
\end{quote}

\texttt{width, height}
\begin{quote}
The width and height of the final plot in inches.
\end{quote}

\texttt{family()}
\begin{quote}
An optional name of the font family to use for the plot. For example, \texttt{family = "Helvetica"} will use the Helvetica font for all text (assuming you have Helvetica on your system). For more help on using different fonts, look at section "Using extra fonts in R" in Chapter XX
\end{quote}

\texttt{dev.off()}
\begin{quote}
This is \textit{not} an argument to \texttt{pdf()} and \texttt{jpeg()}. You just need to execute this code after creating the plot to finish creating the image file (see examples below).
\end{quote}

To use these functions to save files, you need to follow 3 steps

1. Execute the \texttt{pdf()} or \texttt{jpeg()} functions with \texttt{file}, \texttt{width} and \texttt{height} arguments.
2. Execute all your plotting code.
3. Complete the file by executing the command \texttt{dev.off()}. This tells R that you’re done creating the file.

Here’s an example of the three steps.
You’ll notice that after you close the plot with `dev.off()`, you’ll see a message in the prompt like "null device".

Using the command `pdf()` will save the file as a pdf. If you use `jpeg()`, it will be saved as a jpeg.
**Test your R Might! Purdy pictures**

For the following exercises, you’ll use datasets from the `yarrr` package. Make sure to install and load the package

1. The `beardlengths` dataframe contains data on the lengths of beards from 3 different pirate ships. Calculate the average beard length for each ship using `aggregate()`, then create the following barplot:

![Barplot of mean beard length by ship](image)

2. Now using the entire `beardlengths` dataframe, create the following pirateplot:
4. Using the pirates dataset, create the following scatterplot showing the relationship between a pirate’s age and how many parrot’s (s)he has owned (hint: to make the points solid and transparent, use `pch = 16`, and `col = gray(level = .5, alpha = .1)`).
10: Plotting: Part Deux

Advanced colors

Shades of gray with gray()

If you’re a lonely, sexually repressed, 50+ year old housewife, then you might want to stick with shades of gray. If so, the function gray(x) is your answer. gray() is a function that takes a number (or vector of numbers) between 0 and 1 as an argument, and returns a shade of gray (or many shades of gray with a vector input). A value of 1 is equivalent to “white” while 0 is equivalent to “black”. This function is very helpful if you want to create shades of gray depending on the value of a numeric vector. For example, if you had survey data and plotted income on the x-axis and happiness on the y-axis of a scatterplot, you could determine the darkness of each point as a function of a third quantitative variable (such as number of children or amount of travel time to work). I plotted an example of this in Figure 59.

```r
inc <- rnorm(n = 200, mean = 50, sd = 50)
hap <- inc + rnorm(n = 200, mean = 20, sd = 15)
drive <- inc + rnorm(n = 200, mean = 8, sd = 5)
plot(x = inc, y = hap, pch = 16,
col = gray((drive - min(drive)) / max(drive - min(drive)), alpha = .4),
cex = 1.5,
xlab = "income", ylab = "happiness"
)
```

Figure 59: Using the gray() function to easily create shades of gray in plotting symbols based on numerical data.
Pirate Palettes

The `yarr` package comes with several color palettes ready for you to use. The palettes are contained in the `piratepal()` function. To see all the palettes, run the following code:

```r
library("yarr")
piratepal(palette = "all",
plot.result = TRUE)
```

Here are all of the pirate palettes

![Pirate Palettes](image)

Once you find a color palette you like, you can save the colors as a vector by setting the `action` argument to "return", and assigning the result to an object. For example, if I want to use the southpark palette and use them in a plot, I would do the following:

```r
# Save the South Park palette
sp.cols <- piratepal(palette = "southpark")

# Create a blank plot
plot(1, xlim = c(0, 6), ylim = c(0, 2),
     bty = "n", type = "n")

# Add points
points(x = 1:5, y = rep(1, 5),
       pch = c(21, 22, 23, 24, 25),
       main = "South Park Colors",
       bg = sp.cols, # Use the South Park Colors
       col = "white", cex = 3)
```
Color Palettes with the RColorBrewer package

If you use many colors in the same plot, it’s probably a good idea to choose colors that compliment each other. An easy way to select colors that go well together is to use a color palette - a collection of colors known to go well together.

One package that is great for getting (and even creating) palettes is RColorBrewer. Here are some of the palettes in the package. The name of each palette is in the first column, and the colors in each palette are in each row:

```
require("RColorBrewer")
display.brewer.all()
```

To use one of the palettes, execute the function `brewer.pal(n, name)`, where \( n \) is the number of colors you want, and `name` is the name of the palette. For example, to get 4 colors from the color set "Set1", you’d use the code

```
my.colors <- brewer.pal(4, "Set1") # 4 colors from Set1
my.colors
```

```
[[1]] "#E41A1C" "#377EB8" "#4DAF4A" "#984EA3"
```

I know the results look like gibberish, but trust me, R will interpret them as the colors in the palette. Once you store the output of the `brewer.pal()` function as a vector (something like `my.colors`),
you can then use this vector as an argument for the colors in your plot.

**Numerically defined color gradients with colorRamp2**

My favorite way to generate colors that represent numerical data is with the function colorRamp2 in the circlize package (the same package that creates that really cool chordDiagram from Chapter 1). The colorRamp2 function allows you to easily generate shades of colors based on numerical data.

The best way to explain how colorRamp2 works is by giving you an example. Let’s say that you want to want to plot data showing the relationship between the number of drinks someone has on average per week and the resulting risk of some adverse health effect. Further, let’s say you want to color the points as a function of the number of packs of cigarettes per week that person smokes, where a value of 0 packs is colored Blue, 10 packs is Orange, and 30 packs is Red. Moreover, you want the values in between these break points of 0, 10 and 30 to be a mix of the colors. For example, the value of 5 (half way between 0 and 10) should be an equal mix of Blue and Orange.

colorRamp2 allows you to do exactly this. The function has three arguments:

- **breaks**: A vector indicating the break points
- **colors**: A vector of colors corresponding to each value in breaks
- **transparency**: A value between 0 and 1 indicating the transparency (1 means fully transparent)

When you run the function, the function will actually return another function that you can then use to generate colors. Once you store the resulting function as an object (something like my.color.fun) you can then apply this new function on numerical data (in our example, the number of cigarettes someone smokes) to obtain the correct color for each data point.

For example, let’s create the color ramp function for our smoking data points. I’ll use colorRamp2 to create a function that I’ll call smoking.colors which takes a number as an argument, and returns the corresponding color:

```r
smoking.colors <- colorRamp2(breaks = c(0, 15, 30),
                               colors = c("blue", "orange", "red"),
                               transparency = .3)
```

```r
library(\"colorRamp2\")
library(\"circlize\")

# Create Data
drinks <- sample(1:20, size = 100, replace = T)
smokes <- sample(1:30, size = 100, replace = T)
risk <- 1 / (1 + exp(-drinks / 20 + norm(100, mean = 0, sd = 13)))

# Create color function from colorRamp
smoking.colors <- colorRamp2(breaks = c(0, 15, 30),
                               colors = c("blue", "orange", "red"),
                               transparency = .3)

# Set up plot layout
layout(matrix(c(1, 1), nrow = 2, ncol = 1),
        height = c(3.5, 5), width = 6)

# Top Plot
par(mar = c(4, 4, 2, 1))
plot(d, ylim = c(-5, 35), ylim = c(-5, 35),
     type = "n", xlab = "Cigarette Packs",
     ylab = "Risk", pch = 16, main = "colorRamp2 Example")

segments(x0 = c(0, 15, 30),
         y0 = rep(0, 3),
         x1 = c(0, 15, 30),
         y1 = rep(1, 3),
         lty = 2)

points(x = rnorm(10, 16, col = smoking.colors(0:30)))

text(x = c(0, 15, 30), y = rep(2, 3),
     labels = c("Blue", "Orange", "Red"))

# Bottom Plot
par(mar = c(4, 4, 5, 1))
plot(x = drinks, y = risk, col = smoking.colors(smokes),
     pch = 16, cex = 2, main = "Plot of (Made-up) Data",
     xlab = "Drinks", ylab = "Risk")

text(x = "Point color indicates smoking rate", cex = 1.5, side = 3)
```

---

**colorRamp2 Example**

```

```

---

**Plot of (Made-up) Data**

```

```

---

**Point color indicates smoking rate**

```

---

```r

```
To see this function in action, check out the the margin Figure for an example, and check out the help menu ?colorRamp2 for more information and examples.

Stealing any color from your screen with a kuler

One of my favorite tricks for getting great colors in R is to use a color kuler. A color kuler is a tool that allows you to determine the exact RGB values for a color on a screen. For example, let’s say that you wanted to use the exact colors used in the Google logo. To do this, you need to use an app that allows you to pick colors off your computer screen. On a Mac, you can use the program called "Digital Color Meter." If you then move your mouse over the color you want, the software will tell you the exact RGB values of that color. In the image below, you can see me figuring out that the RGB value of the G in Google is R: 19, G: 72, B: 206. Using this method, I figured out the four colors of Google! Check out the margin Figure for the grand result.

Figure 60: Stealing colors from the internet. Not illegal (yet).
**Plot margins**

All plots in R have margins surrounding them that separate the main plotting space from the area where the axes, labels and additional text lie. To visualize how R creates plot margins, look at margin Figure.

You can adjust the size of the margins by specifying a margin parameter using the syntax `par(mar = c(a, b, c, d))` before you execute your first high-level plotting function, where a, b, c and d are the size of the margins on the bottom, left, top, and right of the plot. Let’s see how this works by creating two plots with different margins:

In the plot on the left, I’ll set the margins to 3 on all sides. In the plot on the right, I’ll set the margins to 6 on all sides.

```r
par(mfrow = c(1, 2)) # Put plots next to each other

# First Plot
par(mar = rep(2, 4)) # Set the margin on all sides to 2
plot(1:10)
mtext("Small Margins", side = 3, line = 1, cex = 1.2)

# Second Plot
par(mar = rep(6, 4)) # Set the margin on all sides to 6
plot(1:10)
mtext("Large Margins", side = 3, line = 1, cex = 1.2)
```

You’ll notice that the margins are so small in the first plot that you can’t even see the axis labels, while in the second plot there is plenty (probably too much) white space around the plotting region.

In addition to using the `mar` parameter, you can also specify margin sizes with the `mai` parameter. This acts just like `mar` except that the values for `mai` set the margin size in inches.

```r
par(mfrow = c(1, 2))
x.vals <- rnorm(500)
y.vals <- x.vals + rnorm(500, sd = .5)
plot(x.vals, y.vals, xlim = c(-2, 2), ylim = c(-2, 2),
     main = "", xlab = "", ylab = "", xaxt = "n",
     yaxt = "n", bty = "n", pos = 16, col = gray(0.8, alpha = .2))
axis(1, at = seq(-2, 2, .5), col.axis = gray(0.5), col = gray(0.8))
axis(2, at = seq(-2, 2, .5), col.axis = gray(0.5), col = gray(0.8))
par(cex = 1)
par(mar = rep(6, 4))
plot(1, xlim = c(0, 1), ylim = c(0, 1), type = "n",
     main = "", xlab = "", ylab = "", xaxt = "n", yaxt = "n")
rect(0, 0, 1, 1)
rect(2, 2, .85, .8, lty = 2)
arrow(c(5, 5, 5, .85), c(5, .5, 5, .5),
     c(5, 5, .5, 21), c(5, 5, 21, 21),
     code = 3, length = .1)

text(x = c(5, 5, .89, .89),
     y = c(88, 11, 5, 5),
     labels = c("mar[1]", "mar[2]", "mar[3]", "mar[4]"),
     pos = c(2, 2, 1, 1))

text(7, .9, "This is Margin Text Made with mtext()")
```

The default value for `mar` is c(5.1, 4.1, 4.1, 2.1)
Arranging multiple plots with \texttt{par(mfrow)} and \texttt{layout()} 

R makes it easy to arrange multiple plots in the same plotting space. The most common ways to do this is with the \texttt{par(mfrow)} parameter, and the \texttt{layout()} function. Let’s go over each in turn:

Simple plot layouts with \texttt{par(mfrow)} and \texttt{par(mfcol)}

The \texttt{mfrow} and \texttt{mfcol} parameters allow you to create a matrix of plots in one plotting space. Both parameters take a vector of length two as an argument, corresponding to the number of rows and columns in the resulting plotting matrix. For example, the following code sets up a $3 \times 3$ plotting matrix.

\begin{verbatim}
par(mfrow = c(3, 3)) # Create a 3 x 3 plotting matrix
\end{verbatim}

When you execute this code, you won’t see anything happen. However, when you execute your first high-level plotting command, you’ll see that the plot will show up in the space reserved for the first plot (the top left). When you execute a second high-level plotting command, R will place that plot in the second place in the plotting matrix - either the top middle (if using \texttt{par(mfrow)} or the left middle (if using \texttt{par(mfcol)}). As you continue to add high-level plots, R will continue to fill the plotting matrix.

So what’s the difference between \texttt{par(mfrow)} and \texttt{par(mfcol)}? The only difference is that while \texttt{par(mfrow)} puts sequential plots into the plotting matrix by row, \texttt{par(mfcol)} will fill them by column.

When you are finished using a plotting matrix, be sure to reset the plotting parameter back to its default state:

\begin{verbatim}
par(mfrow = c(1, 1))
\end{verbatim}

If you don’t reset the \texttt{mfrow} parameter, R will continue creating new plotting matrices.

Complex plot layouts with \texttt{layout()}

While \texttt{par(mfrow)} allows you to create matrices of plots, it does not allow you to create plots of different sizes. In order to arrange plots in different sized plotting spaces, you need to use the \texttt{layout()} function. Unlike \texttt{par(mfrow)}, \texttt{layout()} is not a plotting parameter, rather it is a function all on its own. Let’s go through the main arguments of \texttt{layout()}:

\begin{verbatim}
layout(mat, widths, heights)
\end{verbatim}
• **mat**: A matrix indicating the location of the next N figures in the global plotting space. Each value in the matrix must be 0 or a positive integer. R will plot the first plot in the entries of the matrix with 1, the second plot in the entries with 2,...

• **widths**: A vector of values for the widths of the columns of the plotting space.

• **heights**: A vector of values for the heights of the rows of the plotting space.

The `layout()` function can be a bit confusing at first, so I think it’s best to start with an example. Let’s say you want to place histograms next to a scatterplot: Let’s do this using `layout`

We’ll begin by creating the **layout matrix**, this matrix will tell R in which order to create the plots:

```r
layout.matrix <- matrix(c(0, 2, 3, 1), nrow = 2, ncol = 2)
layout.matrix
```

```
  [,1] [,2]
[1,]  0  3
[2,]  2  1
```

Looking at the values of `layout.matrix`, you can see that we’ve told R to put the first plot in the bottom right, the second plot on the bottom left, and the third plot in the top right. Because we put a 0 in the first element, R knows that we don’t plan to put anything in the top left area.

Now, because our layout matrix has two rows and two columns, we need to set the widths and heights of the two columns. We do this using a numeric vector of length 2. I’ll set the heights of the two rows to 1 and 2 respectively, and the widths of the columns to 1 and 2 respectively. Now, when I run the code `layout.show(3)`, R will show us the plotting region we set up (see margin Figure 62).

Now we’re ready to put the plots together:

```r
layout.mat <- matrix(c(2, 1, 0, 3), nrow = 2, ncol = 2)
layout(mat = layout.mat, 
       heights = c(1, 2), # Heights of the two rows 
       widths = c(2, 1)) # Widths of the two columns )
x.vals <- rnorm(100, mean = 100, sd = 10)
y.vals <- x.vals + rnorm(100, mean = 0, sd = 10)
# Plot 1: Scatterplot
par(mar = c(5, 4, 0, 0))
plot(x.vals, y.vals)
```

Figure 62: A plotting layout created by setting a layout matrix and specific heights and widths.
abline(h = median(y.vals), lty = 1, col = "gray")
abline(v = median(x.vals), lty = 1, col = "gray")

# Plot 2: X boxplot
par(mar = c(0, 4, 0, 0))
boxplot(x.vals, xaxt = "n",
       yaxt = "n", bty = "n", yaxt = "n",
       col = "white", frame = F, horizontal = T)

# Plot 3: Y boxplot
par(mar = c(5, 0, 0, 0))
boxplot(y.vals, xaxt = "n",
       yaxt = "n", bty = "n", yaxt = "n",
       col = "white", frame = F)
Additional Tips

- To change the background color of a plot, add the command `par(bg = mycolor)` (where mycolor is the color you want to use) prior to creating the plot. For example, the following code will put a light gray background behind a histogram:

  ```r
  par(bg = gray(.9))
  hist(x = rnorm(100))
  ```

  ![Histogram of rnorm(100)](image)

See Figure 63 for a nicer example.

- Sometimes you’ll mess so much with plotting parameters that you may want to set things back to their default value. To see the default values for all the plotting parameters, execute the code `par()` to print the default parameter values for all plotting parameters to the console.

  ```r
  pdf("/Library/ole2pdf.pdf", width = 6, height = 6)
  par(bg = rgb(0, 0, 0, 0.1))
  hist(x = rnorm(100))
  ```

  ![Histogram of rnorm(100)](image)

Figure 63: Use `par(bg = my.color)` before creating a plot to add a colored background. The design of this plot was inspired by http://www.vox.com/2015/5/20/8625785/expensive-wine-
plot(1, xlab = "Named Colors", ylab = "", xaxt = "n", yaxt = "n")

rect(xleft = rep(1:26, each = 26)[1:length(colors())] - .5,
    ybottom = rep(26:1, times = 26)[1:length(colors())] - .5,
    xright = rep(1:26, each = 26)[1:length(colors())] + .5,
    ytop = rep(26:1, times = 26)[1:length(colors())] + .5,
    col = colors()
)

text(x = rep(1:26, each = 26)[1:length(colors())],
     y = rep(26:1, times = 26)[1:length(colors())],
     labels = colors(), cex = .3
)

Figure 90: The colors stored in colors().

Named Colors