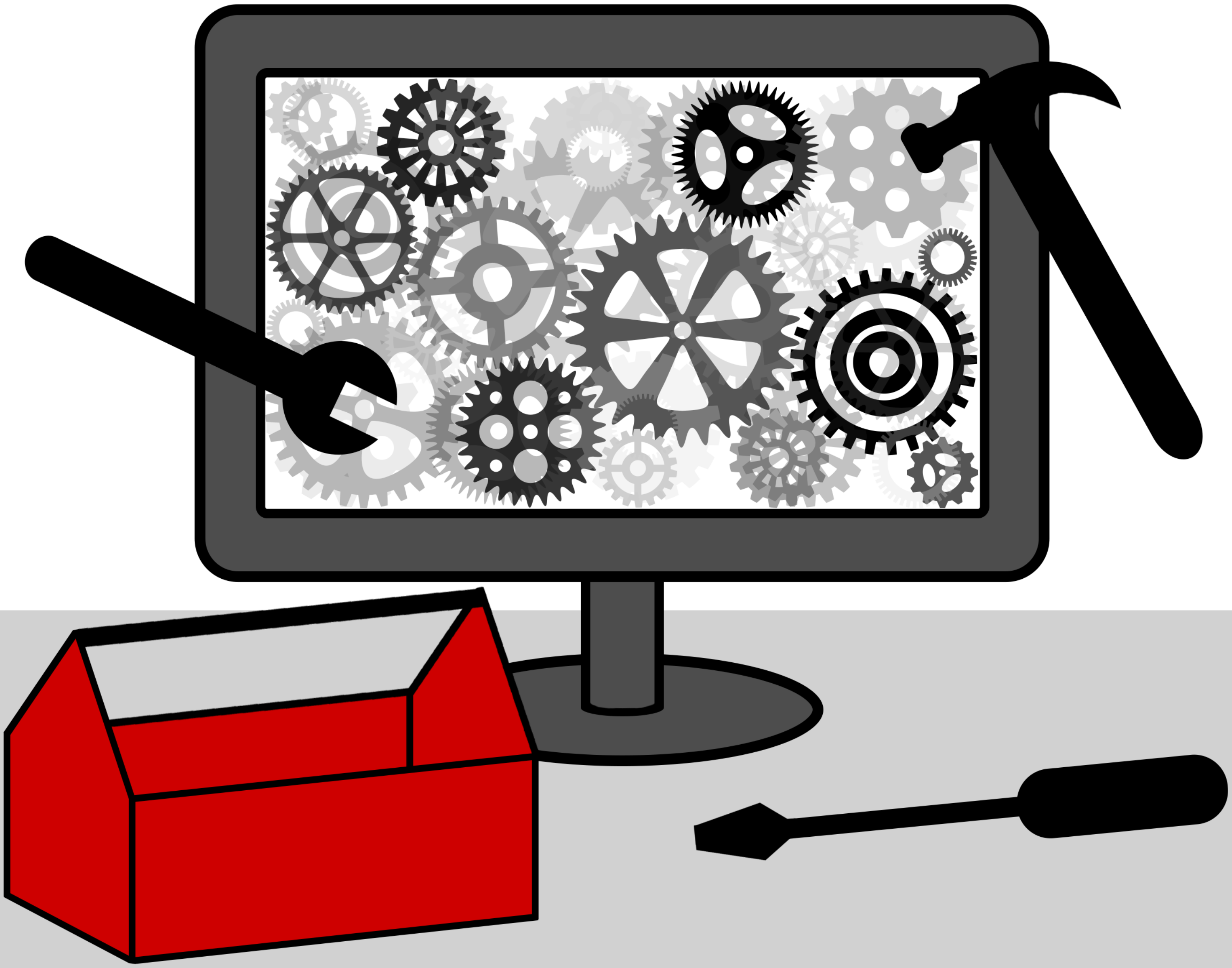
Mastering Software Development in R



Roger D Peng   Sean Kross   Brooke Anderson

Mastering Software Development in R

Roger D. Peng, Sean Kross and Brooke Anderson

This book is for sale at http://leanpub.com/msdr

This version was published on 2016-10-21

Contents

Introduction

⚠ NOTE: This book is under active development.

This book is designed to be used in conjunction with the course sequence *Mastering Software Development in R*, available on Coursera. The book covers R software development for building data science tools. As the field of data science evolves, it has become clear that software development skills are essential for producing useful data science results and products. You will obtain rigorous training in the R language, including the skills for handling complex data, building R packages and developing custom data visualizations. You will learn modern software development practices to build tools that are highly reusable, modular, and suitable for use in a team-based environment or a community of developers.

Setup {-}

This book makes use of the following R packages, which should be installed to take full advantage of the examples.

```
choroplethr
choroplethrMaps
data.table
datasets
dlnm
dplyr
faraway
ggmap
ggplot2
ggthemes
gridExtra
httr
knitr
lubridate
magrittr
methods
microbenchmark
profvis
pryr
purrr
readr
stringr
tidyr
```

```
tidyverse
titanic
```

You can install all of these packages with the following code:

```
install.packages(c("choroplethr", "choroplethrMaps", "data.table", "datasets", "dlnm", "dplyr", "far\
away", "ggmap", "ggplot2", "ggthemes", "gridExtra", "httr", "knitr", "lubridate", "magrittr", "metho\
ds", "microbenchmark", "profvis", "pryr", "purrr", "readr", "stringr", "tidyr", "tidyverse", "titani\
c"))
```

1. The R Programming Environment

This chapter provides a rigorous introduction to the R programming language, with a particular focus on using R for software development in a data science setting. Whether you are part of a data science team or working individually within a community of developers, this chapter will give you the knowledge of R needed to make useful contributions in those settings.

As the first chapter in this book, the chapter provides the essential foundation of R needed for the following chapters. We cover basic R concepts and language fundamentals, key concepts like tidy data and related "tidyverse" tools, processing and manipulation of complex and large datasets, handling textual data, and basic data science tasks. Upon finishing this chapter, you will have fluency at the R console and will be able to create tidy datasets from a wide range of possible data sources.

The learning objectives for this chapter are to:

- Develop fluency in using R at the console
- Execute basic arithmetic operations
- Subset and index R objects
- Remove missing values from an R object
- Modify object attributes and metadata
- Describe differences in different R classes and data types
- Read tabular data into R and read in web data via web scraping tools and APIs
- Define tidy data and to transform non-tidy data into tidy data
- Manipulate and transform a variety of data types, including dates, times, and text data
- Describe how memory is used in R sessions to store R objects
- Read and manipulate large datasets
- Describe how to diagnose programming problems and to look up answers from the web or forums

1.1 Crash Course on R Syntax

Note: Some of the material in this section is taken from R Programming for Data Science.

The learning objectives for this section are to:

- Develop fluency in using R at the console
- Execute basic arithmetic operations
- Subset and index R objects
- Remove missing values from an R object

- Modify object attributes and metadata
- Describe differences in different R classes and data types

At the R prompt we type expressions. The `<-` symbol (*gets arrow*) is the assignment operator.

```
x <- 1
print(x)
[1] 1
x
[1] 1
msg <- "hello"
```

The grammar of the language determines whether an expression is complete or not.

```
x <-  ## Incomplete expression
```

The # character indicates a comment. Anything to the right of the # (including the # itself) is ignored. This is the only comment character in R. Unlike some other languages, R does not support multi-line comments or comment blocks.

Evaluation

When a complete expression is entered at the prompt, it is evaluated and the result of the evaluated expression is returned. The result may be *auto-printed*.

```
x <- 5  ## nothing printed
x       ## auto-printing occurs
[1] 5
print(x)  ## explicit printing
[1] 5
```

The `[1]` shown in the output indicates that `x` is a vector and `5` is its first element.

Typically with interactive work, we do not explicitly print objects with the `print` function; it is much easier to just auto-print them by typing the name of the object and hitting return/enter. However, when writing scripts, functions, or longer programs, there is sometimes a need to explicitly print objects because auto-printing does not work in those settings.

When an R vector is printed you will notice that an index for the vector is printed in square brackets `[]` on the side. For example, see this integer sequence of length 20.

```
x <- 11:30
x
 [1] 11 12 13 14 15 16 17 18 19 20 21 22
[13] 23 24 25 26 27 28 29 30
```

The numbers in the square brackets are not part of the vector itself, they are merely part of the *printed output*.

With R, it's important that one understand that there is a difference between the actual R object and the manner in which that R object is printed to the console. Often, the printed output may have additional bells and whistles to make the output more friendly to the users. However, these bells and whistles are not inherently part of the object.

Note that the `:` operator is used to create integer sequences.

R Objects

R has five basic or "atomic" classes of objects:

- character
- numeric (real numbers)
- integer
- complex
- logical (True/False)

The most basic type of R object is a vector. Empty vectors can be created with the `vector()` function. There is really only one rule about vectors in R, which is: **A vector can only contain objects of the same class**.

But of course, like any good rule, there is an exception, which is a *list*, which we will get to a bit later. A list is represented as a vector but can contain objects of different classes. Indeed, that's usually why we use them.

There is also a class for "raw" objects, but they are not commonly used directly in data analysis and we won't cover them here.

Numbers

Numbers in R are generally treated as numeric objects (i.e. double precision real numbers). This means that even if you see a number like "1" or "2" in R, which you might think of as integers, they are likely represented behind the scenes as numeric objects (so something like "1.00" or "2.00"). This isn't important most of the time…except when it is.

If you explicitly want an integer, you need to specify the `L` suffix. So entering `1` in R gives you a numeric object; entering `1L` explicitly gives you an integer object.

There is also a special number `Inf` which represents infinity. This allows us to represent entities like `1 / 0`. This way, `Inf` can be used in ordinary calculations; e.g. `1 / Inf` is 0.

The value `NaN` represents an undefined value ("not a number"); e.g. 0 / 0; `NaN` can also be thought of as a missing value (more on that later)

Creating Vectors

Watch a video of this section

The `c()` function can be used to create vectors of objects by concatenating things together.

```
x <- c(0.5, 0.6)       ## numeric
x <- c(TRUE, FALSE)    ## logical
x <- c(T, F)           ## logical
x <- c("a", "b", "c")  ## character
x <- 9:29              ## integer
x <- c(1+0i, 2+4i)     ## complex
```

Note that in the above example, `T` and `F` are short-hand ways to specify `TRUE` and `FALSE`. However, in general one should try to use the explicit `TRUE` and `FALSE` values when indicating logical values. The `T` and `F` values are primarily there for when you're feeling lazy.

You can also use the `vector()` function to initialize vectors.

```
x <- vector("numeric", length = 10)
x
 [1] 0 0 0 0 0 0 0 0 0 0
```

Mixing Objects

There are occasions when different classes of R objects get mixed together. Sometimes this happens by accident but it can also happen on purpose. So what happens with the following code?

```
y <- c(1.7, "a")   ## character
y <- c(TRUE, 2)    ## numeric
y <- c("a", TRUE)  ## character
```

In each case above, we are mixing objects of two different classes in a vector. But remember that the only rule about vectors says this is not allowed. When different objects are mixed in a vector, *coercion* occurs so that every element in the vector is of the same class.

In the example above, we see the effect of *implicit coercion*. What R tries to do is find a way to represent all of the objects in the vector in a reasonable fashion. Sometimes this does exactly what you want and…sometimes not. For example, combining a numeric object with a character object will create a character vector, because numbers can usually be easily represented as strings.

Explicit Coercion

Objects can be explicitly coerced from one class to another using the `as.*` functions, if available.

```
x <- 0:6
class(x)
[1] "integer"
as.numeric(x)
[1] 0 1 2 3 4 5 6
as.logical(x)
[1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
as.character(x)
[1] "0" "1" "2" "3" "4" "5" "6"
```

Sometimes, R can't figure out how to coerce an object and this can result in NAs being produced.

```
x <- c("a", "b", "c")
as.numeric(x)
Warning: NAs introduced by coercion
[1] NA NA NA
as.logical(x)
[1] NA NA NA
as.complex(x)
Warning: NAs introduced by coercion
[1] NA NA NA
```

When nonsensical coercion takes place, you will usually get a warning from R.

Matrices

Matrices are vectors with a *dimension* attribute. The dimension attribute is itself an integer vector of length 2 (number of rows, number of columns)

```
m <- matrix(nrow = 2, ncol = 3)
m
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA
dim(m)
[1] 2 3
attributes(m)
$dim
[1] 2 3
```

Matrices are constructed *column-wise*, so entries can be thought of starting in the "upper left" corner and running down the columns.

```
m <- matrix(1:6, nrow = 2, ncol = 3)
m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Matrices can also be created directly from vectors by adding a dimension attribute.

```
m <- 1:10
m
 [1]  1  2  3  4  5  6  7  8  9 10
dim(m) <- c(2, 5)
m
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

Matrices can be created by *column-binding* or *row-binding* with the `cbind()` and `rbind()` functions.

```
x <- 1:3
y <- 10:12
cbind(x, y)
     x  y
[1,] 1 10
[2,] 2 11
[3,] 3 12
rbind(x, y)
  [,1] [,2] [,3]
x    1    2    3
y   10   11   12
```

Lists

Lists are a special type of vector that can contain elements of different classes. Lists are a very important data type in R and you should get to know them well. Lists, in combination with the various "apply" functions discussed later, make for a powerful combination.

Lists can be explicitly created using the `list()` function, which takes an arbitrary number of arguments.

```
x <- list(1, "a", TRUE, 1 + 4i)
x
[[1]]
[1] 1

[[2]]
[1] "a"

[[3]]
[1] TRUE

[[4]]
[1] 1+4i
```

We can also create an empty list of a prespecified length with the `vector()` function

```
x <- vector("list", length = 5)
x
[[1]]
NULL

[[2]]
NULL

[[3]]
NULL

[[4]]
NULL

[[5]]
NULL
```

Factors

Factors are used to represent categorical data and can be unordered or ordered. One can think of a factor as an integer vector where each integer has a *label*. Factors are important in statistical modeling and are treated specially by modelling functions like `lm()` and `glm()`.

Using factors with labels is *better* than using integers because factors are self-describing. Having a variable that has values "Male" and "Female" is better than a variable that has values 1 and 2.

Factor objects can be created with the `factor()` function.

```
x <- factor(c("yes", "yes", "no", "yes", "no"))
x
[1] yes yes no  yes no
Levels: no yes
table(x)
x
 no yes
  2   3
## See the underlying representation of factor
unclass(x)
[1] 2 2 1 2 1
attr(,"levels")
[1] "no"  "yes"
```

Often factors will be automatically created for you when you read a dataset in using a function like `read.table()`. Those functions often, as a default, create factors when they encounter data that look like characters or strings.

The order of the levels of a factor can be set using the `levels` argument to `factor()`. This can be important in linear modelling because the first level is used as the baseline level. This feature can also be used to customize order in plots that include factors, since by default factors are plotted in the order of their levels.

```
x <- factor(c("yes", "yes", "no", "yes", "no"))
x  ## Levels are put in alphabetical order
[1] yes yes no  yes no
Levels: no yes
x <- factor(c("yes", "yes", "no", "yes", "no"),
            levels = c("yes", "no"))
x
[1] yes yes no  yes no
Levels: yes no
```

Missing Values

Missing values are denoted by `NA` or `NaN` for undefined mathematical operations.

- `is.na()` is used to test objects if they are `NA`
- `is.nan()` is used to test for `NaN`
- `NA` values have a class also, so there are integer `NA`, character `NA`, etc.
- A `NaN` value is also `NA` but the converse is not true

```
## Create a vector with NAs in it
x <- c(1, 2, NA, 10, 3)
## Return a logical vector indicating which elements are NA
is.na(x)
[1] FALSE FALSE  TRUE FALSE FALSE
## Return a logical vector indicating which elements are NaN
is.nan(x)
[1] FALSE FALSE FALSE FALSE FALSE
```

```
## Now create a vector with both NA and NaN values
x <- c(1, 2, NaN, NA, 4)
is.na(x)
[1] FALSE FALSE  TRUE  TRUE FALSE
is.nan(x)
[1] FALSE FALSE  TRUE FALSE FALSE
```

Data Frames

Data frames are used to store tabular data in R. They are an important type of object in R and are used in a variety of statistical modeling applications. Hadley Wickham's package dplyr has an optimized set of functions designed to work efficiently with data frames, and `ggplot2` plotting functions work best with data stored in data frames.

Data frames are represented as a special type of list where every element of the list has to have the same length. Each element of the list can be thought of as a column and the length of each element of the list is the number of rows.

Unlike matrices, data frames can store different classes of objects in each column. Matrices must have every element be the same class (e.g. all integers or all numeric).

In addition to column names, indicating the names of the variables or predictors, data frames have a special attribute called `row.names` which indicate information about each row of the data frame.

Data frames are usually created by reading in a dataset using the `read.table()` or `read.csv()`. However, data frames can also be created explicitly with the `data.frame()` function or they can be coerced from other types of objects like lists.

Data frames can be converted to a matrix by calling `data.matrix()`. While it might seem that the `as.matrix()` function should be used to coerce a data frame to a matrix, almost always, what you want is the result of `data.matrix()`.

```
x <- data.frame(foo = 1:4, bar = c(T, T, F, F))
x
  foo   bar
1   1  TRUE
2   2  TRUE
3   3 FALSE
4   4 FALSE
nrow(x)
[1] 4
ncol(x)
[1] 2
```

Names

R objects can have names, which is very useful for writing readable code and self-describing objects. Here is an example of assigning names to an integer vector.

```
x <- 1:3
names(x)
NULL
names(x) <- c("New York", "Seattle", "Los Angeles")
x
   New York     Seattle Los Angeles
          1           2           3
names(x)
[1] "New York"    "Seattle"     "Los Angeles"
```

Lists can also have names, which is often very useful.

```
x <- list("Los Angeles" = 1, Boston = 2, London = 3)
x
$`Los Angeles`
[1] 1

$Boston
[1] 2

$London
[1] 3
names(x)
[1] "Los Angeles" "Boston"      "London"
```

Matrices can have both column and row names.

```
m <- matrix(1:4, nrow = 2, ncol = 2)
dimnames(m) <- list(c("a", "b"), c("c", "d"))
m
  c d
a 1 3
b 2 4
```

Column names and row names can be set separately using the `colnames()` and `rownames()` functions.

```
colnames(m) <- c("h", "f")
rownames(m) <- c("x", "z")
m
  h f
x 1 3
z 2 4
```

Note that for data frames, there is a separate function for setting the row names, the `row.names()` function. Also, data frames do not have column names, they just have names (like lists). So to set the column names of a data frame just use the `names()` function. Yes, I know its confusing. Here's a quick summary:

Object	Set column names	Set row names
data frame	`names()`	`row.names()`
matrix	`colnames()`	`rownames()`

Attributes

In general, R objects can have attributes, which are like metadata for the object. These metadata can be very useful in that they help to describe the object. For example, column names on a data frame help to tell us what data are contained in each of the columns. Some examples of R object attributes are

- names, dimnames
- dimensions (e.g. matrices, arrays)
- class (e.g. integer, numeric)
- length
- other user-defined attributes/metadata

Attributes of an object (if any) can be accessed using the `attributes()` function. Not all R objects contain attributes, in which case the `attributes()` function returns `NULL`.

Summary

There are a variety of different builtin-data types in R. In this section we have reviewed the following

- atomic classes: numeric, logical, character, integer, complex
- vectors, lists
- factors
- missing values
- data frames and matrices

All R objects can have attributes that help to describe what is in the object. Perhaps the most useful attributes are names, such as column and row names in a data frame, or simply names in a vector or list. Attributes like dimensions are also important as they can modify the behavior of objects, like turning a vector into a matrix.

1.2 The Importance of Tidy Data

The learning objectives for this section are to:

- Define tidy data and to transform non-tidy data into tidy data

One unifying concept of this book is the notion of **tidy data**. As defined by Hadley Wickham in his 2014 paper published in the *Journal of Statistical Software*, a tidy dataset has the following properties:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

The purpose of defining tidy data is to highlight the fact that *most data do not start out life as tidy*. In fact, much of the work of data analysis may involve simply making the data tidy (at least this has been our experience). Once a dataset is tidy, it can be used as input into a variety of other functions that may transform, model, or visualize the data.

As a quick example, consider the following data illustrating death rates in Virginia in 1940 in a classic table format:

```
      Rural Male Rural Female Urban Male Urban Female
50-54       11.7          8.7       15.4          8.4
55-59       18.1         11.7       24.3         13.6
60-64       26.9         20.3       37.0         19.3
65-69       41.0         30.9       54.6         35.1
70-74       66.0         54.3       71.1         50.0
```

While this format is canonical and is useful for quickly observing the relationship between multiple variables, it is not tidy. This format violates the tidy form because there are variables in both the rows and columns. In this case the variables are age category, gender, and urban-ness. Finally, the death rate itself, which is the fourth variable, is presented inside the table.

Converting this data to tidy format would give us

```
library(tidyr)
library(dplyr)

VADeaths %>%
  tbl_df() %>%
  mutate(age = row.names(VADeaths)) %>%
  gather(key, death_rate, -age) %>%
  separate(key, c("urban", "gender"), sep = " ") %>%
  mutate(age = factor(age), urban = factor(urban), gender = factor(gender))
# A tibble: 20 × 4
      age  urban gender death_rate
   <fctr> <fctr> <fctr>      <dbl>
1   50-54  Rural   Male       11.7
2   55-59  Rural   Male       18.1
3   60-64  Rural   Male       26.9
4   65-69  Rural   Male       41.0
5   70-74  Rural   Male       66.0
6   50-54  Rural Female        8.7
7   55-59  Rural Female       11.7
8   60-64  Rural Female       20.3
9   65-69  Rural Female       30.9
10  70-74  Rural Female       54.3
11  50-54  Urban   Male       15.4
12  55-59  Urban   Male       24.3
13  60-64  Urban   Male       37.0
14  65-69  Urban   Male       54.6
15  70-74  Urban   Male       71.1
16  50-54  Urban Female        8.4
17  55-59  Urban Female       13.6
18  60-64  Urban Female       19.3
19  65-69  Urban Female       35.1
20  70-74  Urban Female       50.0
```

The "Tidyverse"

There are a number of R packages that take advantage of the tidy data form and can be used to do interesting things with data. Many (but not all) of these packages are written by Hadley Wickham and the collection of packages is sometimes referred to as the "tidyverse" because of their dependence on and presumption of tidy data. "Tidyverse" packages include

- ggplot2: a plotting system based on the grammar of graphics
- magrittr: defines the `%>%` operator for chaining functions together in a series of operations on data
- dplyr: a suite of (fast) functions for working with data frames
- tidyr: easily tidy data with `spread()` and `gather()` functions

We will be using these packages extensively in this book.

The "tidyverse" package can be used to install all of the packages in the tidyverse at once. For example, instead of starting an R script with this:

```
library(dplyr)
library(tidyr)
library(readr)
library(ggplot2)
```

You can start with this:

```
library(tidyverse)
```

1.3 Reading Tabular Data with the `readr` Package

The learning objectives for this section are to:

- Read tabular data into R and read in web data via web scraping tools and APIs

The `readr` package is the primary means by which we will read tablular data, most notably, comma-separated-value (CSV) files. The `readr` package has a few functions in it for reading and writing tabular data—we will focus on the `read_csv` function. The `readr` package is available on CRAN and the code for the package is maintained on GitHub.

The importance of the `read_csv` function is perhaps better understood from an historical perspective. R's built in `read.csv` function similarly reads CSV files, but the `read_csv` function in `readr` builds on that by removing some of the quirks and "gotchas" of `read.csv` as well as dramatically optimizing the speed with which it can read data into R. The `read_csv` function also adds some nice user-oriented features like a progress meter and a compact method for specifying column types.

The only required argument to `read_csv` is a character string specifying the path to the file to read. A typical call to `read_csv` will look as follows.

```
library(readr)
teams <- read_csv("data/team_standings.csv")
Parsed with column specification:
cols(
  Standing = col_integer(),
  Team = col_character()
)
teams
# A tibble: 32 × 2
   Standing        Team
      <int>       <chr>
1         1       Spain
2         2 Netherlands
3         3     Germany
4         4     Uruguay
5         5   Argentina
6         6      Brazil
```

```
7          7       Ghana
8          8    Paraguay
9          9       Japan
10        10       Chile
# ... with 22 more rows
```

By default, `read_csv` will open a CSV file and read it in line-by-line. It will also (by default), read in the first few rows of the table in order to figure out the type of each column (i.e. integer, character, etc.). In the code example above, you can see that `read_csv` has correctly assigned an integer class to the "Standing" variable in the input data and a character class to the "Team" variable. From the `read_csv` help page:

> If [the argument for `col_types` is] 'NULL', all column types will be imputed from the first 1000 rows on the input. This is convenient (and fast), but not robust. If the imputation fails, you'll need to supply the correct types yourself.

You can also specify the type of each column with the `col_types` argument. In general, it's a good idea to specify the column types explicitly. This rules out any possible guessing errors on the part of `read_csv`. Also, specifying the column types explicitly provides a useful safety check in case anything about the dataset should change without you knowing about it.

```
teams <- read_csv("data/team_standings.csv", col_types = "cc")
```

Note that the `col_types` argument accepts a compact representation. Here `"cc"` indicates that the first column is `character` and the second column is `character` (there are only two columns). Using the `col_types` argument is useful because often it is not easy to automatically figure out the type of a column by looking at a few rows (especially if a column has many missing values).

The `read_csv` function will also read compressed files automatically. There is no need to decompress the file first or use the `gzfile` connection function. The following call reads a gzip-compressed CSV file containing download logs from the RStudio CRAN mirror.

```
logs <- read_csv("data/2016-07-19.csv.gz", n_max = 10)
Parsed with column specification:
cols(
  date = col_date(format = ""),
  time = col_time(format = ""),
  size = col_integer(),
  r_version = col_character(),
  r_arch = col_character(),
  r_os = col_character(),
  package = col_character(),
  version = col_character(),
  country = col_character(),
  ip_id = col_integer()
)
```

Note that the message ("Parse with column specification ...") printed after the call indicates that `read_csv` may have had some difficulty identifying the type of each column. This can be solved by using the `col_types` argument.

```
logs <- read_csv("data/2016-07-20.csv.gz", col_types = "ccicccccci", n_max = 10)
logs
# A tibble: 10 × 10
         date     time    size r_version r_arch      r_os      package
        <chr>    <chr>   <int>     <chr>  <chr>     <chr>        <chr>
1  2016-07-20 06:04:55  144723     3.3.1   i386   mingw32       gtools
2  2016-07-20 06:04:51 2049711     3.3.0   i386   mingw32    rmarkdown
3  2016-07-20 06:04:35   26252      <NA>   <NA>      <NA> R.methodsS3
4  2016-07-20 06:04:34  556091    2.13.1 x86_64   mingw32       tibble
5  2016-07-20 06:03:46  313363    2.13.1 x86_64   mingw32    iterators
6  2016-07-20 06:03:47  378892     3.3.1 x86_64   mingw32      foreach
7  2016-07-20 06:04:46   41228     3.3.1 x86_64 linux-gnu      moments
8  2016-07-20 06:04:34  403177      <NA>   <NA>      <NA>         R.oo
9  2016-07-20 06:04:53     525     3.3.0 x86_64 linux-gnu          rgl
10 2016-07-20 06:04:29  755720     3.2.5 x86_64   mingw32    geosphere
# ... with 3 more variables: version <chr>, country <chr>, ip_id <int>
```

You can specify the column type in a more detailed fashion by using the various `col_*` functions. For example, in the log data above, the first column is actually a date, so it might make more sense to read it in as a Date variable. If we wanted to just read in that first column, we could do

```
logdates <- read_csv("data/2016-07-20.csv.gz",
                     col_types = cols_only(date = col_date()),
                     n_max = 10)
logdates
# A tibble: 10 × 1
         date
       <date>
1  2016-07-20
2  2016-07-20
3  2016-07-20
4  2016-07-20
5  2016-07-20
6  2016-07-20
7  2016-07-20
8  2016-07-20
9  2016-07-20
10 2016-07-20
```

Now the `date` column is stored as a `Date` object which can be used for relevant date-related computations (for example, see the `lubridate` package).

The `read_csv` function has a `progress` option that defaults to TRUE. This options provides a nice progress meter while the CSV file is being read. However, if you are using `read_csv` in a function, or perhaps embedding it in a loop, it's probably best to set `progress = FALSE`.

The `readr` package includes a variety of functions in the `read_*` family that allow you to read in data from different formats of flat files. The following table gives a guide to several functions in the `read_*` family.

`readr` function	**Use**
`read_csv`	Reads comma-separated file
`read_csv2`	Reads semicolon-separated file
`read_tsv`	Reads tab-separated file
`read_delim`	General function for reading delimited files
`read_fwf`	Reads fixed width files
`read_log`	Reads log files

1.4 Reading Web-Based Data

The learning objectives for this section are to:

- Read in web data via web scraping tools and APIs

Not only can you read in data locally stored on your computer, with R it is also fairly easy to read in data stored on the web.

Flat files online

The simplest way to do this is if the data is available online as a flat file (see note below). For example, the "Extended Best Tracks" for the North Atlantic are hurricane tracks that include both the best estimate of the central location of each storm and also gives estimates of how far winds of certain speeds extended from the storm's center in four quadrants of the storm (northeast, northwest, southeast, southwest) at each measurement point. You can see this file online here.

How can you tell if you've found a flat file online? Here are a couple of clues:

- It will not have any formatting. Instead, it will look online as if you opened a file in a text editor on your own computer.
- It will often have a web address that ends with a typical flat file extension (`".csv"`, `".txt"`, or `".fwf"`, for example).

Here are a couple of examples of flat files available online:

- Population mean county centers for Colorado counties, from the US Census
- Weather in Fort Collins, Colorado, in 2015, from Weather Underground

If you copy and paste the web address for this file, you'll see that the url for this example hurricane data file is non-secure (starts with `http:`) and that it ends with a typical flat file extension (`.txt`, in this case). You can read this file into your R session using the same `readr` function that you would use to read it in if the file were stored on your computer.

First, you can create an R object with the filepath to the file. In the case of online files, that's the url. To fit the long web address comfortably in an R script window, you can use the `paste0` function to paste pieces of the web address together:

```
ext_tracks_file <- paste0("http://rammb.cira.colostate.edu/research/",
                          "tropical_cyclones/tc_extended_best_track_dataset/",
                          "data/ebtrk_atlc_1988_2015.txt")
```

Next, since this web-based file is a fixed width file, you'll need to define the width of each column, so that R will know where to split between columns. You can then use the `read_fwf` function from the `readr` package to read the file into your R session. This data, like a lot of weather data, uses the string `"-99"` for missing data, and you can specify that missing value character with the `na` argument in `read_fwf`. Also, the online file does not include column names, so you'll have to use the data documentation file for the dataset to determine and set those yourself.

```
library(readr)

# Create a vector of the width of each column
ext_tracks_widths <- c(7, 10, 2, 2, 3, 5, 5, 6, 4, 5, 4, 4, 5, 3, 4, 3, 3, 3,
                       4, 3, 3, 3, 4, 3, 3, 3, 2, 6, 1)

# Create a vector of column names, based on the online documentation for this data
ext_tracks_colnames <- c("storm_id", "storm_name", "month", "day",
                          "hour", "year", "latitude", "longitude",
                          "max_wind", "min_pressure", "rad_max_wind",
                          "eye_diameter", "pressure_1", "pressure_2",
                          paste("radius_34", c("ne", "se", "sw", "nw"), sep = "_"),
                          paste("radius_50", c("ne", "se", "sw", "nw"), sep = "_"),
                          paste("radius_64", c("ne", "se", "sw", "nw"), sep = "_"),
                          "storm_type", "distance_to_land", "final")

# Read the file in from its url
ext_tracks <- read_fwf(ext_tracks_file,
                       fwf_widths(ext_tracks_widths, ext_tracks_colnames),
                       na = "-99")
```

```
ext_tracks[1:3, 1:9]
# A tibble: 3 × 9
  storm_id storm_name month   day  hour  year latitude longitude max_wind
     <chr>      <chr> <chr> <chr> <chr> <int>    <dbl>     <dbl>    <int>
1   AL0188    ALBERTO    08    05    18  1988     32.0      77.5       20
2   AL0188    ALBERTO    08    06    00  1988     32.8      76.2       20
3   AL0188    ALBERTO    08    06    06  1988     34.0      75.2       20
```

For some fixed width files, you may be able to save the trouble of counting column widths by using the `fwf_empty` function in the `readr` package. This function guesses the widths of columns based on the positions of empty columns. However, the example hurricane dataset we are using here is a bit too messy for this– in some cases, there are values from different columns that are not separated by white space. Just as it is typically safer for you to specify column types yourself, rather than relying on R to correctly guess them, it is also safer when reading in a fixed width file to specify column widths yourself.

You can use some `dplyr` functions to check out the dataset once it's in R (there will be much more about `dplyr` in the next section). For example, the following call prints a sample of four rows of data from Hurricane Katrina, with, for each row, the date and time, maximum wind speed, minimum pressure, and the radius of maximum winds of the storm for that observation:

```
library(dplyr)

ext_tracks %>%
  filter(storm_name == "KATRINA") %>%
  select(month, day, hour, max_wind, min_pressure, rad_max_wind) %>%
  sample_n(4)
# A tibble: 4 × 6
  month   day  hour max_wind min_pressure rad_max_wind
  <chr> <chr> <chr>    <int>        <int>        <int>
1    08    25    12       55          994           15
2    08    24    18       40         1003           55
3    08    25    00       45         1000           55
4    08    28    12      145          909           20
```

With the functions in the `readr` package, you can also read in flat files from secure urls (ones that starts with `https:`). (This is not true with the `read.table` family of functions from base R.) One example where it is common to find flat files on secure sites is on GitHub. If you find a file with a flat file extension in a GitHub repository, you can usually click on it and then choose to view the "Raw" version of the file, and get to the flat file version of the file.

For example, the CDC Epidemic Prediction Initiative has a GitHub repository with data on Zika cases, including files on cases in Brazil. When we wrote this, the most current file was available here, with the raw version (i.e., a flat file) available by clicking the "Raw" button on the top right of the first site.

```
zika_file <- paste0("https://raw.githubusercontent.com/cdcepi/zika/master/",
                    "Brazil/COES_Microcephaly/data/COES_Microcephaly-2016-06-25.csv")
zika_brazil <- read_csv(zika_file)

zika_brazil %>%
  select(location, value, unit)
# A tibble: 210 × 3
                  location value  unit
                     <chr> <int> <chr>
1              Brazil-Acre     2 cases
2           Brazil-Alagoas    75 cases
3             Brazil-Amapa     7 cases
4          Brazil-Amazonas     8 cases
5             Brazil-Bahia   263 cases
6             Brazil-Ceara   124 cases
7  Brazil-Distrito_Federal     5 cases
8    Brazil-Espirito_Santo    13 cases
9             Brazil-Goias    14 cases
10         Brazil-Maranhao   131 cases
# ... with 200 more rows
```

Requesting data through a web API

Web APIs are growing in popularity as a way to access open data from government agencies, companies, and other organizations. “API” stands for “Application Program Interface””; an API provides the rules for software applications to interact. In the case of open data APIs, they provide the rules you need to know to write R code to request and pull data from the organization’s web server into your R session. Usually, some of the computational burden of querying and subsetting the data is taken on by the source’s server, to create the subset of requested data to pass to your computer. In practice, this means you can often pull the subset of data you want from a very large available dataset without having to download the full dataset and load it locally into your R session.

As an overview, the basic steps for accessing and using data from a web API when working in R are:

- Figure out the API rules for HTTP requests
- Write R code to create a request in the proper format
- Send the request using GET or POST HTTP methods
- Once you get back data from the request, parse it into an easier-to-use format if necessary

To get the data from an API, you should first read the organization’s API documentation. An organization will post details on what data is available through their API(s), as well as how to set up HTTP requests to get that data– to request the data through the API, you will typically need to send the organization’s web server an HTTP request using a GET or POST method. The API documentation details will typically show an example GET or POST request for the API, including the base URL to use and the possible query parameters that can be used to customize the dataset request.

For example, the National Aeronautics and Space Administration (NASA) has an API for pulling the Astronomy Picture of the Day. In their API documentation, they specify that the base URL for the API request should be “https://api.nasa.gov/planetary/apod” and that you can include parameters to specify the date of the daily picture you want, whether to pull a high-resolution version of the picture, and a NOAA API key you have requested from NOAA.

Many organizations will require you to get an API key and use this key in each of your API requests. This key allows the organization to control API access, including enforcing rate limits per user. API rate limits restrict how often you can request data (e.g., an hourly limit of 1,000 requests per user for NASA APIs).

API keys should be kept private, so if you are writing code that includes an API key, be very careful not to include the actual key in any code made public (including any code in public GitHub repositories). One way to do this is to save the value of your key in a file named `.Renviron` in your home directory. This file should be a plain text file and must end in a blank line. Once you’ve saved your API key to a global variable in that file (e.g., with a line added to the `.Renviron` file like `NOAA_API_KEY="abdafjsiopnab038"`), you can assign the key value to an R object in an R session using the `Sys.getenv` function (e.g., `noaa_api_key <- Sys.getenv("NOAA_API_KEY")`), and then use this object (`noaa_api_key`) anywhere you would otherwise have used the character string with your API key.

To find more R packages for accessing and exploring open data, check out the Open Data CRAN task view. You can also browse through the ROpenSci packages, all of which have GitHub repositories where you can further explore how each package works. ROpenSci is an organization with the mission to create open software tools for science. If you create your own package to access data relevant to scientific research through an API, consider submitting it for peer-review through ROpenSci.

The `riem` package, developed by Maelle Salmon and an ROpenSci package, is an excellent and straightforward example of how you can use R to pull open data through a web API. This package allows you to pull weather data from airports around the world directly from the Iowa Environmental Mesonet. To show you how to pull data into R through an API, in this section we will walk you through code in the `riem` package or code based closely on code in the package.

To get a certain set of weather data from the Iowa Environmental Mesonet, you can send an HTTP request specifying a base URL, “https://mesonet.agron.iastate.edu/cgi-bin/request/asos.py/”, as well as some parameters describing the subset of dataset you want (e.g., date ranges, weather variables, output format). Once you know the rules for the names and possible values of these parameters (more on that below), you can submit an HTTP GET request using the `GET` function from the `httr` package.

When you are making an HTTP request using the `GET` or `POST` functions from the `httr` package, you can include the key-value pairs for any query parameters as a list object in the `query` argurment of the function. For example, suppose you want to get wind speed in miles per hour (`data = "sped"`) for Denver, CO, (`station = "DEN"`) for the month of June 2016 (`year1 = "2016"`, `month1 = "6"`, etc.) in Denver's local time zone (`tz = "America/Denver"`) and in a comma-separated file (`format = "comma"`). To get this weather dataset, you can run:

```
library(httr)
meso_url <- "https://mesonet.agron.iastate.edu/cgi-bin/request/asos.py/"
denver <- GET(url = meso_url,
                    query = list(station = "DEN",
                                 data = "sped",
                                 year1 = "2016",
                                 month1 = "6",
                                 day1 = "1",
                                 year2 = "2016",
                                 month2 = "6",
                                 day2 = "30",
                                 tz = "America/Denver",
                                 format = "comma")) %>%
  content() %>%
  read_csv(skip = 5, na = "M")

denver %>% slice(1:3)
# A tibble: 3 × 3
  station               valid  sped
    <chr>              <dttm> <dbl>
1     DEN 2016-06-01 00:00:00   9.2
2     DEN 2016-06-01 00:05:00   9.2
3     DEN 2016-06-01 00:10:00   6.9
```

The `content` call in this code extracts the content from the response to the HTTP request sent by the `GET` function. The Iowa Environmental Mesonet API offers the option to return the requested data in a comma-separated file (`format = "comma"` in the GET request), so here `content` and `read_csv` are used to extract and read in that csv file. Usually, data will be returned in a JSON format instead. We include more details later in this section on parsing data returned in a JSON format.

The only tricky part of this process is figuring out the available parameter names (e.g., `station`) and possible values for each (e.g., `"DEN"` for Denver). Currently, the details you can send in an HTTP request through Iowa Environmental Mesonet's API include:

- A four-character weather station identifier (`station`)
- The weather variables (e.g., temperature, wind speed) to include (`data`)
- Starting and ending dates describing the range for which you'd like to pull data (`year1`, `month1`, `day1`, `year2`, `month2`, `day2`)
- The time zone to use for date-times for the weather observations (`tz`)

- Different formatting options (e.g., delimiter to use in the resulting data file [`format`], whether to include longitude and latitude)

Typically, these parameter names and possible values are explained in the API documentation. In some cases, however, the documentation will be limited. In that case, you may be able to figure out possible values, especially if the API specifies a GET rather than POST method, by playing around with the website's point-and-click interface and then looking at the url for the resulting data pages. For example, if you look at the Iowa Environmental Mesonet's page for accessing this data, you'll notice that the point-and-click web interface allows you the options in the list above, and if you click through to access a dataset using this interface, the web address of the data page includes these parameter names and values.

The `riem` package implements all these ideas in three very clean and straightforward functions. You can explore the code behind this package and see how these ideas can be incorporated into a small R package, in the `/R` directory of the package's GitHub page.

R packages already exist for many open data APIs. If an R package already exists for an API, you can use functions from that package directly, rather than writing your own code using the API protocols and `httr` functions. Other examples of existing R packages to interact with open data APIs include:

- `twitteR`: Twitter
- `rnoaa`: National Oceanic and Atmospheric Administration
- `Quandl`: Quandl (financial data)
- `RGoogleAnalytics`: Google Analytics
- `censusr`, `acs`: United States Census
- `WDI`, `wbstats`: World Bank
- `GuardianR`, `rdian`: The Guardian Media Group
- `blsAPI`: Bureau of Labor Statistics
- `rtimes`: New York Times
- `dataRetrieval`, `waterData`: United States Geological Survey

If an R package doesn't exist for an open API and you'd like to write your own package, find out more about writing API packages with this vignette for the httr package. This document includes advice on error handling within R code that accesses data through an open API.

Scraping web data

You can also use R to pull and clean web-based data that is not accessible through a web API or as an online flat file. In this case, the strategy will often be to pull in the full web page file (often in HTML or XML) and then parse or clean it within R.

The `rvest` package is a good entry point for handling more complex collection and cleaning of web-based data. This package includes functions, for example, that allow you to select certain elements from the code for a web page (e.g., using the `html_node` and `xml_node` functions), to parse tables in an HTML document into R data frames (`html_table`), and to parse, fill out, and submit HTML forms (`html_form`, `set_values`, `submit_form`). Further details on web scraping with R are beyond the scope of this course, but if you're interested, you can find out more through the rvest GitHub README.

Parsing JSON, XML, or HTML data

Often, data collected from the web, including the data returned from an open API or obtained by scraping a web page, will be in JSON, XML, or HTML format. To use data in a JSON, XML, or HTML format in R, you need to parse the file from its current format and convert it into an R object more useful for analysis.

Typically, JSON-, XML-, or HTML-formatted data is parsed into a list in R, since list objects allow for a lot of flexibility in the structure of the data. However, if the data is structured appropriately, you can often parse data into another type of object (a data frame, for example, if the data fits well into a two-dimensional format of rows and columns). If the data structure of the data that you are pulling in is complex but consistent across different observations, you may alternatively want to create a custom object type to parse the data into.

There are a number of packages for parsing data from these formats, including `jsonlite` and `xml2`. To find out more about parsing data from typical web formats, and for more on working with web-based documents and data, see the CRAN task view for Web Technologies and Services

1.5 Basic Data Manipulation

The learning objectives for this section are to:

- Transform non-tidy data into tidy data
- Manipulate and transform a variety of data types, including dates, times, and text data

The two packages `dplyr` and `tidyr`, both "tidyverse" packages, allow you to quickly and fairly easily clean up your data. These packages are not very old, and so much of the example R code you might find in books or online might not use the functions we use in examples in this section (although this is quickly changing for new books and for online examples). Further, there are many people who are used to using R base functionality to clean up their data, and some of them still do not use these packages much when cleaning data. We think, however, that `dplyr` is easier for people new to R to learn than learning how to clean up data using base R functions, and we also think it produces code that is much easier to read, which is useful in maintaining and sharing code.

For many of the examples in this section, we will use the `ext_tracks` hurricane dataset we input from a url as an example in a previous section of this book. If you need to load a version of that data, we have also saved it locally, so you can create an R object with the example data for this section by running:

```
ext_tracks_file <- "data/ebtrk_atlc_1988_2015.txt"
ext_tracks_widths <- c(7, 10, 2, 2, 3, 5, 5, 6, 4, 5, 4, 4, 5, 3, 4, 3, 3, 3,
                       4, 3, 3, 3, 4, 3, 3, 3, 2, 6, 1)
ext_tracks_colnames <- c("storm_id", "storm_name", "month", "day",
                          "hour", "year", "latitude", "longitude",
                          "max_wind", "min_pressure", "rad_max_wind",
                          "eye_diameter", "pressure_1", "pressure_2",
                          paste("radius_34", c("ne", "se", "sw", "nw"), sep = "_"),
                          paste("radius_50", c("ne", "se", "sw", "nw"), sep = "_"),
                          paste("radius_64", c("ne", "se", "sw", "nw"), sep = "_"),
                          "storm_type", "distance_to_land", "final")
ext_tracks <- read_fwf(ext_tracks_file,
                       fwf_widths(ext_tracks_widths, ext_tracks_colnames),
                       na = "-99")
```

Piping

The `dplyr` and `tidyr` functions are often used in conjunction with piping, which is done with the `%>%` function from the `magrittr` package. Piping can be done with many R functions, but is especially common with `dplyr` and `tidyr` functions. The concept is straightforward– the pipe passes the data frame output that results from the function right before the pipe to input it as the first argument of the function right after the pipe.

Here is a generic view of how this works in code, for a pseudo-function named `function` that inputs a data frame as its first argument:

```
# Without piping
function(dataframe, argument_2, argument_3)

# With piping
dataframe %>%
  function(argument_2, argument_3)
```

For example, without piping, if you wanted to see the time, date, and maximum winds for Katrina from the first three rows of the `ext_tracks` hurricane data, you could run:

```
katrina <- filter(ext_tracks, storm_name == "KATRINA")
katrina_reduced <- select(katrina, month, day, hour, max_wind)
head(katrina_reduced, 3)
# A tibble: 3 × 4
  month   day  hour max_wind
  <chr> <chr> <chr>    <int>
1    10    28    18       30
2    10    29    00       30
3    10    29    06       30
```

In this code, you are creating new R objects at each step, which makes the code cluttered and also requires copying the data frame several times into memory. As an alternative, you could just wrap one function inside another:

```
head(select(filter(ext_tracks, storm_name == "KATRINA"),
            month, day, hour, max_wind), 3)
# A tibble: 3 × 4
  month   day  hour max_wind
  <chr> <chr> <chr>    <int>
1    10    28    18       30
2    10    29    00       30
3    10    29    06       30
```

This avoids re-assigning the data frame at each step, but quickly becomes ungainly, and it's easy to put arguments in the wrong layer of parentheses. Piping avoids these problems, since at each step you can send the output from the last function into the next function as that next function's first argument:

```
ext_tracks %>%
  filter(storm_name == "KATRINA") %>%
  select(month, day, hour, max_wind) %>%
  head(3)
# A tibble: 3 × 4
  month   day  hour max_wind
  <chr> <chr> <chr>    <int>
1    10    28    18       30
2    10    29    00       30
3    10    29    06       30
```

Summarizing data

The `dplyr` and `tidyr` packages have numerous functions (sometimes referred to as "verbs") for cleaning up data. We'll start with the functions to summarize data.

The primary of these is `summarize`, which inputs a data frame and creates a new data frame with the requested summaries. In conjunction with `summarize`, you can use other functions from `dplyr` (e.g., `n`, which counts the number of observations in a given column) to create this summary. You can also use R functions from other packages or base R functions to create the summary.

For example, say we want a summary of the number of observations in the `ext_tracks` hurricane dataset, as well as the highest measured maximum windspeed (given by the column `max_wind` in the dataset) in any of the storms, and the lowest minimum pressure (`min_pressure`). To create this summary, you can run:

```
ext_tracks %>%
  summarize(n_obs = n(),
            worst_wind = max(max_wind),
            worst_pressure = min(min_pressure))
# A tibble: 1 × 3
  n_obs worst_wind worst_pressure
  <int>      <int>          <int>
1 11824        160              0
```

This summary provides particularly useful information for this example data, because it gives an unrealistic value for minimum pressure (0 hPa). This shows that this dataset will need some cleaning. The highest wind speed observed for any of the storms, 160 knots, is more reasonable.

You can also use `summarize` with functions you've written yourself, which gives you a lot of power in summarizing data in interesting ways. As a simple example, if you wanted to present the maximum wind speed in the summary above using miles per hour rather than knots, you could write a function to perform the conversion, and then use that function within the `summarize` call:

```
knots_to_mph <- function(knots){
  mph <- 1.152 * knots
}

ext_tracks %>%
  summarize(n_obs = n(),
            worst_wind = knots_to_mph(max(max_wind)),
            worst_pressure = min(min_pressure))
# A tibble: 1 × 3
  n_obs worst_wind worst_pressure
  <int>      <dbl>          <int>
1 11824     184.32              0
```

So far, we've only used `summarize` to create a single-line summary of the data frame. In other words, the summary functions are applied across the entire dataset, to return a single value for each summary statistic. However, often you might want summaries stratified by a certain grouping characteristic of the data. For the hurricane data, for example, you might want to get the worst wind and worst pressure by storm, rather than across all storms.

You can do this by grouping your data frame by one of its column variables, using the function `group_by`, and then using `summarize`. The `group_by` function does not make a visible change to a data frame, although you can see, if you print out a grouped data frame, that the new grouping variable will be listed under "Groups" at the top of a print-out:

```
ext_tracks %>%
  group_by(storm_name, year) %>%
  head()
Source: local data frame [6 x 29]
Groups: storm_name, year [1]

  storm_id storm_name month   day  hour  year latitude longitude max_wind
     <chr>      <chr> <chr> <chr> <chr> <int>    <dbl>     <dbl>    <int>
1   AL0188    ALBERTO    08    05    18  1988     32.0      77.5       20
2   AL0188    ALBERTO    08    06    00  1988     32.8      76.2       20
3   AL0188    ALBERTO    08    06    06  1988     34.0      75.2       20
4   AL0188    ALBERTO    08    06    12  1988     35.2      74.6       25
5   AL0188    ALBERTO    08    06    18  1988     37.0      73.5       25
6   AL0188    ALBERTO    08    07    00  1988     38.7      72.4       25
# ... with 20 more variables: min_pressure <int>, rad_max_wind <int>,
#   eye_diameter <int>, pressure_1 <int>, pressure_2 <int>,
#   radius_34_ne <int>, radius_34_se <int>, radius_34_sw <int>,
#   radius_34_nw <int>, radius_50_ne <int>, radius_50_se <int>,
#   radius_50_sw <int>, radius_50_nw <int>, radius_64_ne <int>,
#   radius_64_se <int>, radius_64_sw <int>, radius_64_nw <int>,
#   storm_type <chr>, distance_to_land <int>, final <chr>
```

As a note, since hurricane storm names repeat at regular intervals until they are retired, to get a separate summary for each unique storm, this example requires grouping by both `storm_name` and `year`.

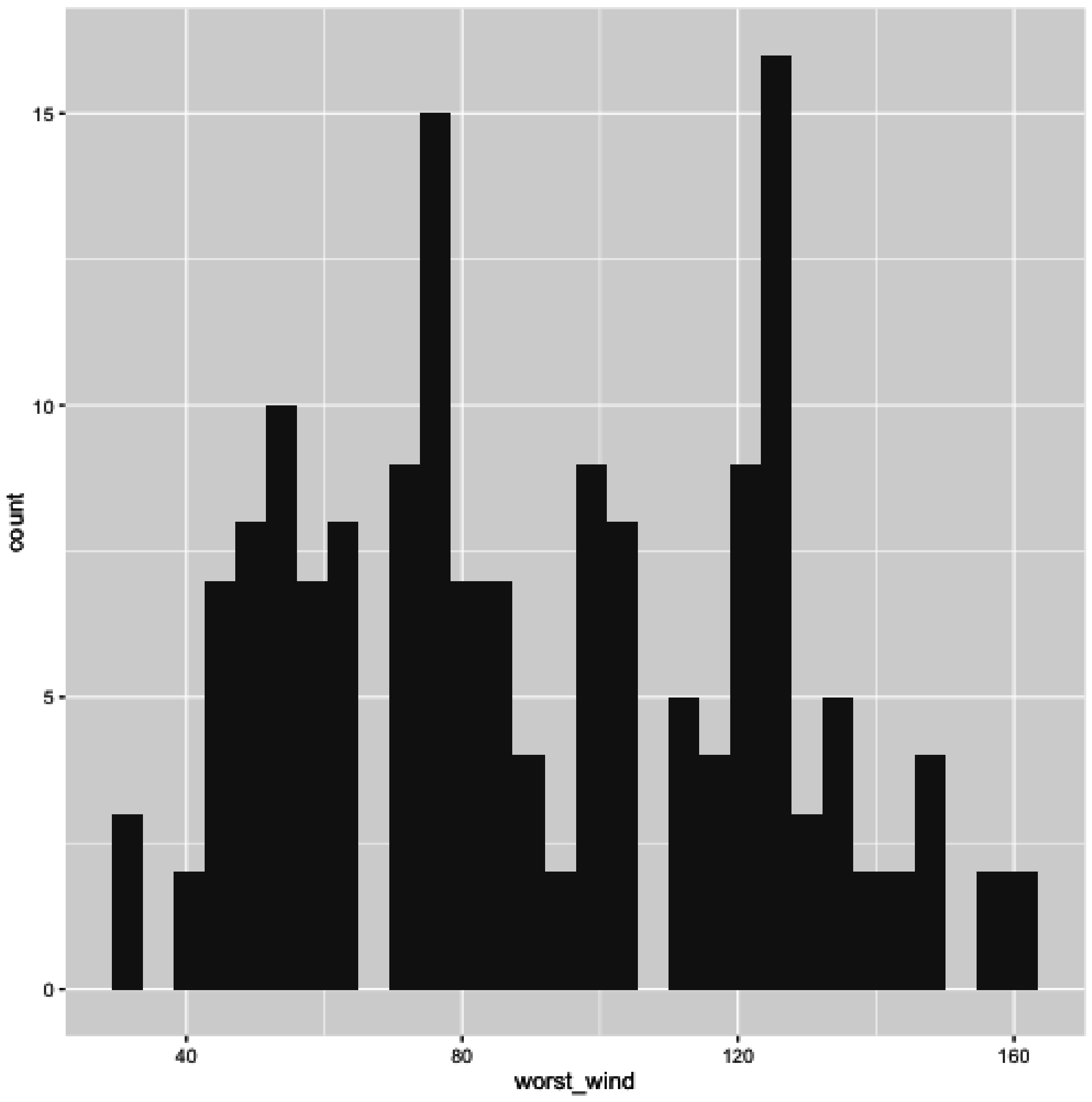
Even though applying the `group_by` function does not cause a noticeable change to the data frame itself, you'll notice the difference in grouped and ungrouped data frames when you use `summarize` on the data frame. If a data frame is grouped, all summaries are calculated and given separately for each unique value of the grouping variable:

```
ext_tracks %>%
  group_by(storm_name, year) %>%
  summarize(n_obs = n(),
            worst_wind = max(max_wind),
            worst_pressure = min(min_pressure))
Source: local data frame [378 x 5]
Groups: storm_name [?]

   storm_name  year n_obs worst_wind worst_pressure
        <chr> <int> <int>      <int>          <int>
1     ALBERTO  1988    13         35           1002
2     ALBERTO  1994    31         55            993
3     ALBERTO  2000    87        110            950
4     ALBERTO  2006    37         60            969
5     ALBERTO  2012    20         50            995
6        ALEX  1998    26         45           1002
7        ALEX  2004    25        105            957
8        ALEX  2010    30         90            948
9     ALLISON  1989    28         45            999
```

```
10    ALLISON  1995    33         65            982
# ... with 368 more rows
```

This grouping / summarizing combination can be very useful for quickly plotting interesting summaries of a dataset. For example, to plot a histogram of maximum wind speed observed for each storm (Figure @ref(fig:windhistogram)), you could run:

```
library(ggplot2)
ext_tracks %>%
  group_by(storm_name) %>%
  summarize(worst_wind = max(max_wind)) %>%
  ggplot(aes(x = worst_wind)) + geom_histogram()
```



Histogram of the maximum wind speed observed during a storm for all Atlantic basin tropical storms, 1988–2015.

> We will show a few basic examples of plotting using `ggplot2` functions in this chapter of the book. We will cover plotting much more thoroughly in a later section of the specialization.

From Figure @ref(fig:windhistogram), we can see that only two storms had maximum wind speeds at or above 160 knots (we'll check this later with some other `dplyr` functions).

You cannot make changes to a variable that is being used to group a dataframe. If you try, you will get the error `Error: cannot modify grouping variable`. If you get this error, use the `ungroup` function to remove grouping within a data frame, and then you will be able to mutate any of the variables in the data.

Selecting and filtering data

When cleaning up data, you will need to be able to create subsets of the data, by selecting certain columns or filtering down to certain rows. These actions can be done using the `dplyr` functions `select` and `filter`.

The `select` function subsets certain columns of a data frame. The most basic way to use `select` is select certain columns by specifying their full column names. For example, to select the storm name, date, time, latitude, longitude, and maximum wind speed from the `ext_tracks` dataset, you can run:

```
ext_tracks %>%
  select(storm_name, month, day, hour, year, latitude, longitude, max_wind)
# A tibble: 11,824 × 8
   storm_name month   day  hour  year latitude longitude max_wind
        <chr> <chr> <chr> <chr> <int>    <dbl>     <dbl>    <int>
1     ALBERTO    08    05    18  1988     32.0      77.5       20
2     ALBERTO    08    06    00  1988     32.8      76.2       20
3     ALBERTO    08    06    06  1988     34.0      75.2       20
4     ALBERTO    08    06    12  1988     35.2      74.6       25
5     ALBERTO    08    06    18  1988     37.0      73.5       25
6     ALBERTO    08    07    00  1988     38.7      72.4       25
7     ALBERTO    08    07    06  1988     40.0      70.8       30
8     ALBERTO    08    07    12  1988     41.5      69.0       35
9     ALBERTO    08    07    18  1988     43.0      67.5       35
10    ALBERTO    08    08    00  1988     45.0      65.5       35
# ... with 11,814 more rows
```

There are several functions you can use with `select` that give you more flexibility, and so allow you to select columns without specifying the full names of each column. For example, the `starts_with` function can be used within a `select` function to pick out all the columns that start with a certain text string. As an example of using `starts_with` in conjunction with `select`, in the `ext_tracks` hurricane data, there are a number of columns that say how far from the storm center winds of certain speeds extend. Tropical storms often have asymmetrical wind fields, so these wind radii are given for each quadrant of the storm (northeast, southeast, northwest, and southeast of the storm's center). All of the columns with the radius to which winds of 34 knots or more extend start with "radius_34". To get a dataset with storm names, location, and radii of winds of 34 knots, you could run:

```
ext_tracks %>%
  select(storm_name, latitude, longitude, starts_with("radius_34"))
# A tibble: 11,824 × 7
   storm_name latitude longitude radius_34_ne radius_34_se radius_34_sw
        <chr>    <dbl>     <dbl>        <int>        <int>        <int>
1     ALBERTO     32.0      77.5            0            0            0
2     ALBERTO     32.8      76.2            0            0            0
3     ALBERTO     34.0      75.2            0            0            0
4     ALBERTO     35.2      74.6            0            0            0
5     ALBERTO     37.0      73.5            0            0            0
6     ALBERTO     38.7      72.4            0            0            0
7     ALBERTO     40.0      70.8            0            0            0
8     ALBERTO     41.5      69.0          100          100           50
9     ALBERTO     43.0      67.5          100          100           50
10    ALBERTO     45.0      65.5           NA           NA           NA
# ... with 11,814 more rows, and 1 more variables: radius_34_nw <int>
```

Other functions that can be used with `select` in a similar way include:

- `ends_with`: Select all columns that end with a certain string (for example, `select(ext_tracks, ends_with("ne"))` to get all the wind radii for the northeast quadrant of a storm for the hurricane example data)
- `contains`: Select all columns that include a certain string (`select(ext_tracks, contains("34"))` to get all wind radii for 34-knot winds)
- `matches`: Select all columns that match a certain relative expression (`select(ext_tracks, matches("_[0-9][0-9]_"))` to get all columns where the column name includes two numbers between two underscores, a pattern that matches all of the wind radii columns)

While `select` picks out certain columns of the data frame, `filter` picks out certain rows. With `filter`, you can specify certain conditions using R's logical operators, and the function will return rows that meet those conditions.

R's logical operators include:

Operator	Meaning	Example
`==`	Equals	`storm_name == KATRINA`
`!=`	Does not equal	`min_pressure != 0`
`>`	Greater than	`latitude > 25`
`>=`	Greater than or equal to	`max_wind >= 160`
`<`	Less than	`min_pressure < 900`
`<=`	Less than or equal to	`distance_to_land <= 0`
`%in%`	Included in	`storm_name %in% c("KATRINA", "ANDREW")`
`is.na()`	Is a missing value	`is.na(radius_34_ne)`

If you are ever unsure of how to write a logical statement, but know how to write its opposite, you can use the `!` operator to negate the whole statement. For example, if you wanted to get all storms *except* those named “KATRINA” and “ANDREW”, you could use `!(storm_name %in% c("KATRINA", "ANDREW"))`. A common use of this is to identify observations with non-missing data (e.g., `!(is.na(radius_34_ne))`).

A logical statement, run by itself on a vector, will return a vector of the same length with `TRUE` every time the condition is met and `FALSE` every time it is not.

```
head(ext_tracks$hour)
[1] "18" "00" "06" "12" "18" "00"
head(ext_tracks$hour == "00")
[1] FALSE  TRUE FALSE FALSE FALSE  TRUE
```

When you use a logical statement within `filter`, it will return just the rows where the logical statement is true:

```
ext_tracks %>%
  select(storm_name, hour, max_wind) %>%
  head(9)
# A tibble: 9 × 3
  storm_name  hour max_wind
       <chr> <chr>    <int>
1    ALBERTO    18       20
2    ALBERTO    00       20
3    ALBERTO    06       20
4    ALBERTO    12       25
5    ALBERTO    18       25
6    ALBERTO    00       25
7    ALBERTO    06       30
8    ALBERTO    12       35
9    ALBERTO    18       35

ext_tracks %>%
  select(storm_name, hour, max_wind) %>%
  filter(hour == "00") %>%
  head(3)
# A tibble: 3 × 3
  storm_name  hour max_wind
       <chr> <chr>    <int>
1    ALBERTO    00       20
2    ALBERTO    00       25
3    ALBERTO    00       35
```

Filtering can also be done after summarizing data. For example, to determine which storms had maximum wind speed equal to or above 160 knots, run:

```
ext_tracks %>%
  group_by(storm_name, year) %>%
  summarize(worst_wind = max(max_wind)) %>%
  filter(worst_wind >= 160)
Source: local data frame [2 x 3]
Groups: storm_name [2]

  storm_name  year worst_wind
       <chr> <int>      <int>
1    GILBERT  1988        160
2      WILMA  2005        160
```

If you would like to string several logical conditions together and select rows where all or any of the conditions are true, you can use the "and" (&) or "or" (|) operators. For example, to pull out observations for Hurricane Andrew when it was at or above Category 5 strength (137 knots or higher), you could run:

```
ext_tracks %>%
  select(storm_name, month, day, hour, latitude, longitude, max_wind) %>%
  filter(storm_name == "ANDREW" & max_wind >= 137)
# A tibble: 2 × 7
  storm_name month   day  hour latitude longitude max_wind
       <chr> <chr> <chr> <chr>    <dbl>     <dbl>    <int>
1     ANDREW    08    23    12     25.4      74.2      145
2     ANDREW    08    23    18     25.4      75.8      150
```

Some common errors that come up when using logical operators in R are:

- If you want to check that two things are equal, make sure you use double equal signs (==), not a single one. At best, a single equals sign won't work; in some cases, it will cause a variable to be re-assigned (= can be used for assignment, just like <-).
- If you are trying to check if one thing is equal to one of several things, use %in% rather than ==. For example, if you want to filter to rows of ext_tracks with storm names of "KATRINA" and "ANDREW", you need to use storm_name %in% c("KATRINA", "ANDREW"), not storm_name == c("KATRINA", "ANDREW").
- If you want to identify observations with missing values (or without missing values), you must use the is.na function, not == or !=. For example, is.na(radius_34_ne) will work, but radius_34_ne == NA will not.

Adding, changing, or renaming columns

The mutate function in dplyr can be used to add new columns to a data frame or change existing columns in the data frame. As an example, I'll use the worldcup dataset from the package faraway, which statistics from the 2010 World Cup. To load this example data frame, you can run:

```
library(faraway)
data(worldcup)
```

This dataset has observations by player, including the player's team, position, amount of time played in this World Cup, and number of shots, passes, tackles, and saves. This dataset is currently not tidy, as it has one of the variables (players' names) as rownames, rather than as a column of the data frame. You can use the `mutate` function to move the player names to its own column:

```
worldcup <- worldcup %>%
  mutate(player_name = rownames(worldcup))

worldcup %>% slice(1:3)
     Team   Position Time Shots Passes Tackles Saves player_name
1 Algeria Midfielder   16     0      6       0     0      Abdoun
2   Japan Midfielder  351     0    101      14     0         Abe
3  France   Defender  180     0     91       6     0      Abidal
```

You can also use `mutate` in coordination with `group_by` to create new columns that give summaries within certain windows of the data. For example, the following code will add a column with the average number of shots for a player's position added as a new column. While this code is summarizing the original data to generate the values in this column, `mutate` will add these repeated summary values to the original dataset by group, rather than returning a dataframe with a single row for each of the grouping variables (try replacing `mutate` with `summarize` in this code to make sure you understand the difference).

```
worldcup <- worldcup %>%
  group_by(Position) %>%
  mutate(ave_shots = mean(Shots)) %>%
  ungroup()

worldcup %>% slice(1:3)
# A tibble: 3 × 9
     Team   Position  Time Shots Passes Tackles Saves player_name
   <fctr>     <fctr> <int> <int>  <int>   <int> <int>       <chr>
1 Algeria Midfielder    16     0      6       0     0      Abdoun
2   Japan Midfielder   351     0    101      14     0         Abe
3  France   Defender   180     0     91       6     0      Abidal
# ... with 1 more variables: ave_shots <dbl>
```

If there is a column that you want to rename, but not change, you can use the `rename` function. For example:

```
worldcup %>%
  rename(Name = player_name) %>%
  slice(1:3)
# A tibble: 3 × 9
     Team   Position  Time Shots Passes Tackles Saves   Name ave_shots
   <fctr>     <fctr> <int> <int>  <int>   <int> <int>  <chr>     <dbl>
1 Algeria Midfielder    16     0      6       0     0 Abdoun  2.394737
2   Japan Midfielder   351     0    101      14     0    Abe  2.394737
3  France   Defender   180     0     91       6     0 Abidal  1.164894
```

Spreading and gathering data

The `tidyr` package includes functions to transfer a data frame between *long* and *wide*. Wide format data tends to have different attributes or variables describing an observation placed in separate columns. Long format data tends to have different attributes encoded as levels of a single variable, followed by another column that contains tha values of the observation at those different levels.

In the section on tidy data, we showed an example that used `gather` to convert data into a tidy format. The data is first in an untidy format:

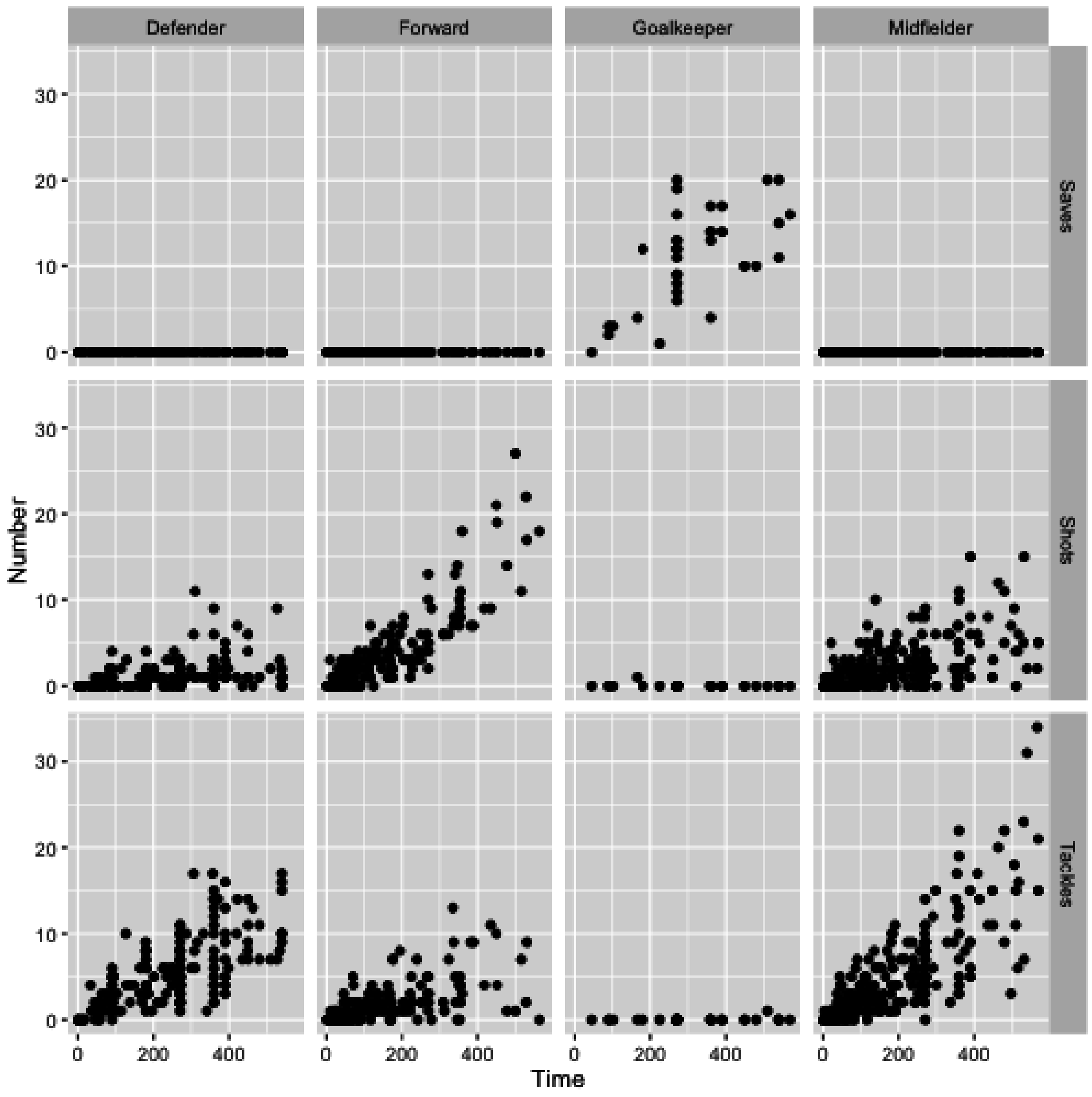
```
data("VADeaths")
head(VADeaths)
      Rural Male Rural Female Urban Male Urban Female
50-54       11.7          8.7       15.4          8.4
55-59       18.1         11.7       24.3         13.6
60-64       26.9         20.3       37.0         19.3
65-69       41.0         30.9       54.6         35.1
70-74       66.0         54.3       71.1         50.0
```

After changing the age categories from row names to a variable (which can be done with the `mutate` function), the key problem with the tidyness of the data is that the variables of urban / rural and male / female are not in their own columns, but rather are embedded in the structure of the columns. To fix this, you can use the `gather` function to gather values spread across several columns into a single column, with the column names gathered into a "key" column. When gathering, exclude any columns that you don't want "gathered" (`age` in this case) by including the column names with a the minus sign in the `gather` function. For example:

```
data("VADeaths")
library(tidyr)

# Move age from row names into a column
VADeaths  <- VADeaths %>%
  tbl_df() %>%
  mutate(age = row.names(VADeaths))
VADeaths
# A tibble: 5 × 5
  `Rural Male` `Rural Female` `Urban Male` `Urban Female`   age
         <dbl>          <dbl>        <dbl>          <dbl> <chr>
1         11.7            8.7         15.4            8.4 50-54
2         18.1           11.7         24.3           13.6 55-59
3         26.9           20.3         37.0           19.3 60-64
4         41.0           30.9         54.6           35.1 65-69
5         66.0           54.3         71.1           50.0 70-74

# Gather everything but age to tidy data
VADeaths %>%
  gather(key = key, value = death_rate, -age)
# A tibble: 20 × 3
     age          key death_rate
   <chr>        <chr>      <dbl>
1  50-54   Rural Male       11.7
2  55-59   Rural Male       18.1
3  60-64   Rural Male       26.9
4  65-69   Rural Male       41.0
5  70-74   Rural Male       66.0
6  50-54 Rural Female        8.7
7  55-59 Rural Female       11.7
8  60-64 Rural Female       20.3
9  65-69 Rural Female       30.9
10 70-74 Rural Female       54.3
11 50-54   Urban Male       15.4
12 55-59   Urban Male       24.3
13 60-64   Urban Male       37.0
14 65-69   Urban Male       54.6
15 70-74   Urban Male       71.1
16 50-54 Urban Female        8.4
17 55-59 Urban Female       13.6
18 60-64 Urban Female       19.3
19 65-69 Urban Female       35.1
20 70-74 Urban Female       50.0
```

Even if your data is in a tidy format, `gather` is occasionally useful for pulling data together to take advantage of faceting, or plotting separate plots based on a grouping variable. For example, if you'd like to plot the relationship between the time a player played in the World Cup and his number of saves, tackles, and shots, with a separate graph for each position (Figure @ref(fig:facetworldcup)), you can use `gather` to pull all the numbers of saves, tackles, and shots into a single column (`Number`) and then use faceting to plot them as separate graphs:

```
library(tidyr)
library(ggplot2)
worldcup %>%
  select(Position, Time, Shots, Tackles, Saves) %>%
  gather(Type, Number, -Position, -Time) %>%
  ggplot(aes(x = Time, y = Number)) +
  geom_point() +
  facet_grid(Type ~ Position)
```



Example of a faceted plot created by taking advantage of the `gather` function to pull together data.

The spread function is less commonly needed to tidy data. It can, however, be useful for creating summary tables. For example, if you wanted to print a table of the average number and range of passes by position for the top four teams in this World Cup (Spain, Netherlands, Uruguay, and Germany), you could run:

```
library(knitr)

# Summarize the data to create the summary statistics you want
wc_table <- worldcup %>%
  filter(Team %in% c("Spain", "Netherlands", "Uruguay", "Germany")) %>%
  select(Team, Position, Passes) %>%
  group_by(Team, Position) %>%
  summarize(ave_passes = mean(Passes),
            min_passes = min(Passes),
            max_passes = max(Passes),
            pass_summary = paste0(round(ave_passes), " (",
                                  min_passes, ", ",
                                  max_passes, ")")) %>%
  select(Team, Position, pass_summary)
# What the data looks like before using `spread`
wc_table
Source: local data frame [16 x 3]
Groups: Team [4]

          Team   Position   pass_summary
        <fctr>     <fctr>          <chr>
1      Germany   Defender  190 (44, 360)
2      Germany    Forward    90 (5, 217)
3      Germany Goalkeeper    99 (99, 99)
4      Germany Midfielder   177 (6, 423)
5  Netherlands   Defender  182 (30, 271)
6  Netherlands    Forward   97 (12, 248)
7  Netherlands Goalkeeper 149 (149, 149)
8  Netherlands Midfielder  170 (22, 307)
9        Spain   Defender   213 (1, 402)
10       Spain    Forward   77 (12, 169)
11       Spain Goalkeeper    67 (67, 67)
12       Spain Midfielder  212 (16, 563)
13     Uruguay   Defender   83 (22, 141)
14     Uruguay    Forward   100 (5, 202)
15     Uruguay Goalkeeper    75 (75, 75)
16     Uruguay Midfielder   100 (1, 252)

# Use spread to create a prettier format for a table
wc_table %>%
  spread(Position, pass_summary) %>%
  kable()
```

Team	Defender	Forward	Goalkeeper	Midfielder
Germany	190 (44, 360)	90 (5, 217)	99 (99, 99)	177 (6, 423)
Netherlands	182 (30, 271)	97 (12, 248)	149 (149, 149)	170 (22, 307)
Spain	213 (1, 402)	77 (12, 169)	67 (67, 67)	212 (16, 563)
Uruguay	83 (22, 141)	100 (5, 202)	75 (75, 75)	100 (1, 252)

Notice in this example how `spread` has been used at the very end of the code sequence to convert the summarized data into a shape that offers a better tabular presentation for a report. In the `spread` call, you first specify the name of the column to use for the new column names (`Position` in this example) and then specify the column to use for the cell values (`pass_summary` here).

In this code, I've used the `kable` function from the `knitr` package to create the summary table in a table format, rather than as basic R output. This function is very useful for formatting basic tables in R markdown documents. For more complex tables, check out the `pander` and `xtable` packages.

Merging datasets

Often, you will have data in two separate datasets that you'd like to combine based on a common variable or variables. For example, for the World Cup example data we've been using, it would be interesting to add in a column with the final standing of each player's team. We've included data with that information in a file called "team_standings.csv", which can be read into the R object `team_standings` with the call:

```
team_standings <- read_csv("data/team_standings.csv")
team_standings %>% slice(1:3)
# A tibble: 3 × 2
  Standing        Team
     <int>       <chr>
1        1       Spain
2        2 Netherlands
3        3     Germany
```

This data frame has one observation per team, and the team names are consistent with the team names in the `worldcup` data frame.

You can use the different functions from the `*_join` family to merge this team standing data with the player statistics in the `worldcup` data frame. Once you've done that, you can use other data cleaning tools from `dplyr` to quickly pull and explore interesting parts of the dataset. The main arguments for the `*_join` functions are the object names of the two data frames to join and `by`, which specifies which variables to use to match up observations from the two dataframes.

There are several functions in the `*_join` family. These functions all merge together two data frames; they differ in how they handle observations that exist in one but not both data frames. Here are the four functions from this family that you will likely use the most often:

Function	What it includes in merged data frame
`left_join`	Includes all observations in the left data frame, whether or not there is a match in the right data frame
`right_join`	Includes all observations in the right data frame, whether or not there is a match in the left data frame
`inner_join`	Includes only observations that are in both data frames
`full_join`	Includes all observations from both data frames

In this table, the "left" data frame refers to the first data frame input in the `*_join` call, while the "right" data frame refers to the second data frame input into the function. For example, in the call

```
left_join(world_cup, team_standings, by = "Team")
```

the `world_cup` data frame is the "left" data frame and the `team_standings` data frame is the "right" data frame. Therefore, using `left_join` would include all rows from `world_cup`, whether or not the player had a team listed in `team_standings`, while `right_join` would include all the rows from `team_standings`, whether or not there were any players from that team in `world_cup`.

Remember that if you are using piping, the first data frame ("left" for these functions) is by default the dataframe created by the code right before the pipe. When you merge data frames as a step in piped code, therefore, the "left" data frame is the one piped into the function while the "right" data frame is the one stated in the `*_join` function call.

As an example of merging, say you want to create a table of the top 5 players by shots on goal, as well as the final standing for each of these player's teams, using the `worldcup` and `team_standings` data. You can do this by running:

```
data(worldcup)
worldcup %>%
  mutate(Name = rownames(worldcup),
         Team = as.character(Team)) %>%
  select(Name, Position, Shots, Team) %>%
  arrange(desc(Shots)) %>%
  slice(1:5) %>%
  left_join(team_standings, by = "Team") %>% # Merge in team standings
  rename("Team Standing" = Standing) %>%
  kable()
```

Name	Position	Shots	Team	Team Standing
212	Forward	27	Ghana	7
560	Forward	22	Spain	1
370	Forward	21	Argentina	5
514	Forward	19	Uruguay	4
174	Forward	18	Uruguay	4

In addition to the merging in this code, there are a few other interesting things to point out:

- The code uses the `as.character` function within a `mutate` call to change the team name from a factor to a character in the `worldcup` data frame. When merging two data frames, it's safest if the column you're using to merge has the same class in each data frame. The "Team" column is a character class in the `team_standings` data frame but a factor class in the `worldcup` data frame, so this call converts that column to a character class in `worldcup`. The `left_join` function will still perform a merge if you don't include this call, but it will throw a warning that it is coercing the column in `worldcup` to a character vector. It's generally safer to do this yourself explictly.
- It uses the `select` function both to remove columns we're not interested in and also to put the columns we want to keep in the order we'd like for the final table.
- It uses `arrange` followed by `slice` to pull out the top 5 players and order them by number of shots.
- For one of the column names, we want to use "Team Standing" rather than the current column name of "Standing". This code uses `rename` at the very end to make this change right before creating the table. You can also use the `col.names` argument in the `kable` function to customize all the column names in the final table, but this `rename` call is a quick fix since we just want to change one column name.

1.6 Working with Dates, Times, Time Zones

The learning objectives for this section are to:

- Transform non-tidy data into tidy data
- Manipulate and transform a variety of data types, including dates, times, and text data

R has special object classes for dates and date-times. It is often worthwhile to convert a column in a data frame to one of these special object types, because you can do some very useful things with date or date-time objects, including pull out the month or day of the week from the observations in the object, or determine the time difference between two values.

Many of the examples in this section use the `ext_tracks` object loaded earlier in the book. If you need to reload that, you can use the following code to do so:

```
ext_tracks_file <- "data/ebtrk_atlc_1988_2015.txt"
ext_tracks_widths <- c(7, 10, 2, 2, 3, 5, 5, 6, 4, 5, 4, 4, 5, 3, 4, 3, 3, 3,
                       4, 3, 3, 3, 4, 3, 3, 3, 2, 6, 1)
ext_tracks_colnames <- c("storm_id", "storm_name", "month", "day",
                          "hour", "year", "latitude", "longitude",
                          "max_wind", "min_pressure", "rad_max_wind",
                          "eye_diameter", "pressure_1", "pressure_2",
                          paste("radius_34", c("ne", "se", "sw", "nw"), sep = "_"),
                          paste("radius_50", c("ne", "se", "sw", "nw"), sep = "_"),
                          paste("radius_64", c("ne", "se", "sw", "nw"), sep = "_"),
                          "storm_type", "distance_to_land", "final")
ext_tracks <- read_fwf(ext_tracks_file,
                       fwf_widths(ext_tracks_widths, ext_tracks_colnames),
                       na = "-99")
```

Converting to a date or date-time class

The `lubridate` package (another package from the "tidyverse") has some excellent functions for working with dates in R. First, this package includes functions to transform objects into date or date-time classes. For example, the `ymd_hm` function (along with other functions in the same family: `ymd`, `ymd_h`, and `ymd_hms`) can be used to convert a vector from character class to R's data and datetime classes, POSIXlt and POSIXct, respectively.

Functions in this family can be used to parse character strings into dates, regardless of how the date is formatted, as long as the date is in the order: year, month, day (and, for time values, hour, minute). For example:

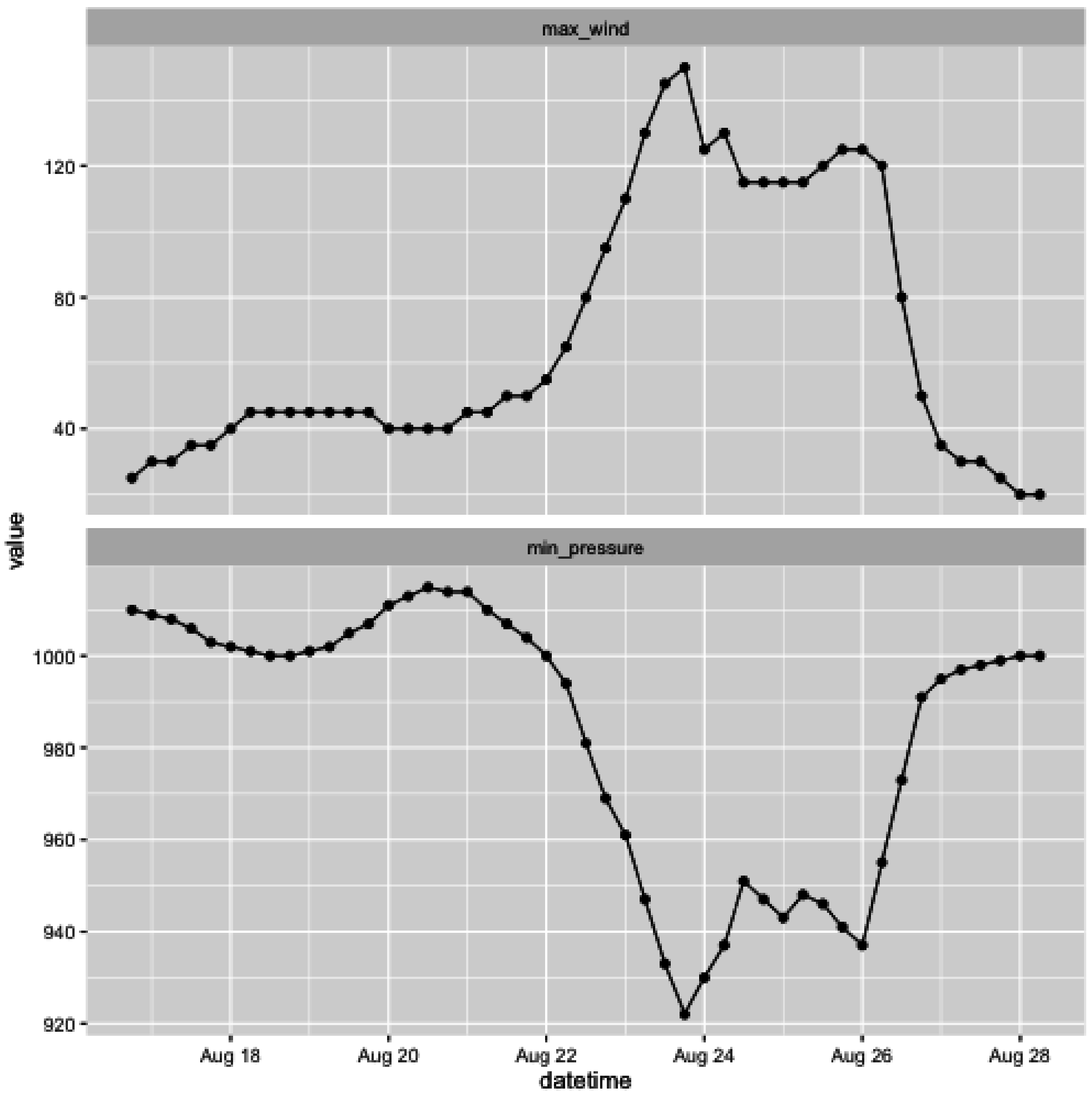
```
library(lubridate)

ymd("2006-03-12")
[1] "2006-03-12"
ymd("'06 March 12")
[1] "2006-03-12"
ymd_hm("06/3/12 6:30 pm")
[1] "2006-03-12 18:30:00 UTC"
```

The following code shows how to use the `ymd_h` function to transform the date and time information in a subset of the hurricane example data called `andrew_tracks` (the storm tracks for Hurricane Andrew) to a date-time class (`POSIXct`). This code also uses the `unite` function from the `tidyr` package to join together date components that were originally in separate columns before applying `ymd_h`.

```
library(dplyr)
library(tidyr)

andrew_tracks <- ext_tracks %>%
  filter(storm_name == "ANDREW" & year == "1992") %>%
  select(year, month, day, hour, max_wind, min_pressure) %>%
  unite(datetime, year, month, day, hour) %>%
  mutate(datetime = ymd_h(datetime))

head(andrew_tracks, 3)
# A tibble: 3 × 3
             datetime max_wind min_pressure
               <dttm>    <int>        <int>
1 1992-08-16 18:00:00       25         1010
2 1992-08-17 00:00:00       30         1009
3 1992-08-17 06:00:00       30         1008
class(andrew_tracks$datetime)
[1] "POSIXct" "POSIXt"
```

Now that the `datetime` variable in this dataset has been converted to a date-time class, the variable becomes much more useful. For example, if you plot a time series using `datetime`, `ggplot2` can recognize that this object is a date-time and will make sensible axis labels. The following code plots maximum wind speed and minimum air pressure at different observation times for Hurricane Andrew (Figure @ref(fig:andrewwind))– check the axis labels to see how they've been formatted. Note that this code uses `gather` from the `tidyr` package to enable easy faceting, to create separate plots for wind speed and air pressure.

```
andrew_tracks %>%
  gather(measure, value, -datetime) %>%
  ggplot(aes(x = datetime, y = value)) +
  geom_point() + geom_line() +
  facet_wrap(~ measure, ncol = 1, scales = "free_y")
```



Example of how variables in a date-time class can be parsed for sensible axis labels.

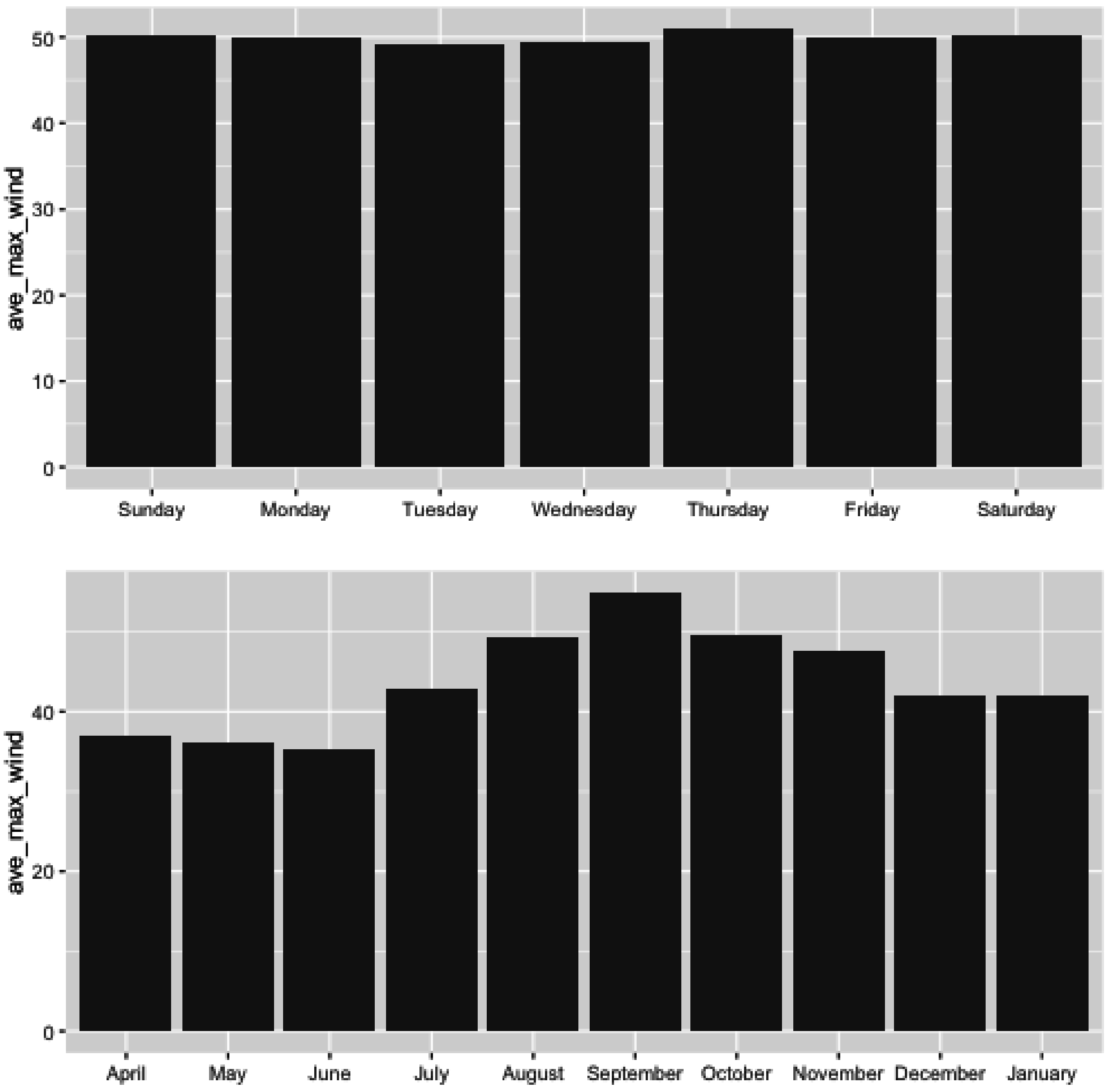
Pulling out date and time elements

Once an object is in a date or date-time class (`POSIXlt` or `POSIXct`, respectively), there are other functions in the `lubridate` package you can use to pull certain elements out of it. For example, you can use the functions `year`, `months`, `mday`, `wday`, `yday`, `weekdays`, `hour`, `minute`, and `second` to pull the year, month, month day, etc., of the date. The following code uses the `datetime` variable in the Hurricane Andrew track data to add new columns for the year, month, weekday, year day, and hour of each observation:

```
andrew_tracks %>%
  select(datetime) %>%
  mutate(year = year(datetime),
         month = months(datetime),
         weekday = weekdays(datetime),
         yday = yday(datetime),
         hour = hour(datetime)) %>%
  slice(1:3)
# A tibble: 3 × 6
             datetime  year  month weekday  yday  hour
               <dttm> <dbl>  <chr>   <chr> <dbl> <int>
1 1992-08-16 18:00:00  1992 August  Sunday   229    18
2 1992-08-17 00:00:00  1992 August  Monday   230     0
3 1992-08-17 06:00:00  1992 August  Monday   230     6
```

This functionality makes it easy to look at patterns in the `max_wind` value by different time groupings, like weekday and month. For example, the following code puts together some of the `dplyr` and `tidyr` data cleaning tools and `ggplot2` plotting functions with these `lubridate` functions to look at the average value of `max_wind` storm observations by day of the week and by month (Figure @ref(fig:stormbytimegroups)).

```
check_tracks <- ext_tracks %>%
  select(month, day, hour, year, max_wind) %>%
  unite(datetime, year, month, day, hour) %>%
  mutate(datetime = ymd_h(datetime),
         weekday = weekdays(datetime),
         weekday = factor(weekday, levels = c("Sunday", "Monday",
                                              "Tuesday", "Wednesday",
                                              "Thursday", "Friday",
                                              "Saturday")),
         month = months(datetime),
         month = factor(month, levels = c("April", "May", "June",
                                          "July", "August", "September",
                                          "October", "November",
                                          "December", "January")))

check_weekdays <- check_tracks %>%
  group_by(weekday) %>%
  summarize(ave_max_wind = mean(max_wind)) %>%
  rename(grouping = weekday)
check_months <- check_tracks %>%
  group_by(month) %>%
  summarize(ave_max_wind = mean(max_wind)) %>%
  rename(grouping = month)

a <- ggplot(check_weekdays, aes(x = grouping, y = ave_max_wind)) +
  geom_bar(stat = "identity") + xlab("")
b <- a %+% check_months

library(gridExtra)
grid.arrange(a, b, ncol = 1)
```



Example of using `lubridate` functions to explore data with a date variable by different time groupings

Based on Figure @ref(fig:stormbytimegroups), there's little pattern in storm intensity by day of the week, but there is a pattern by month, with the highest average wind speed measurements in observations in September and neighboring months (and no storm observations in February or March).

There are a few other interesting things to note about this code:

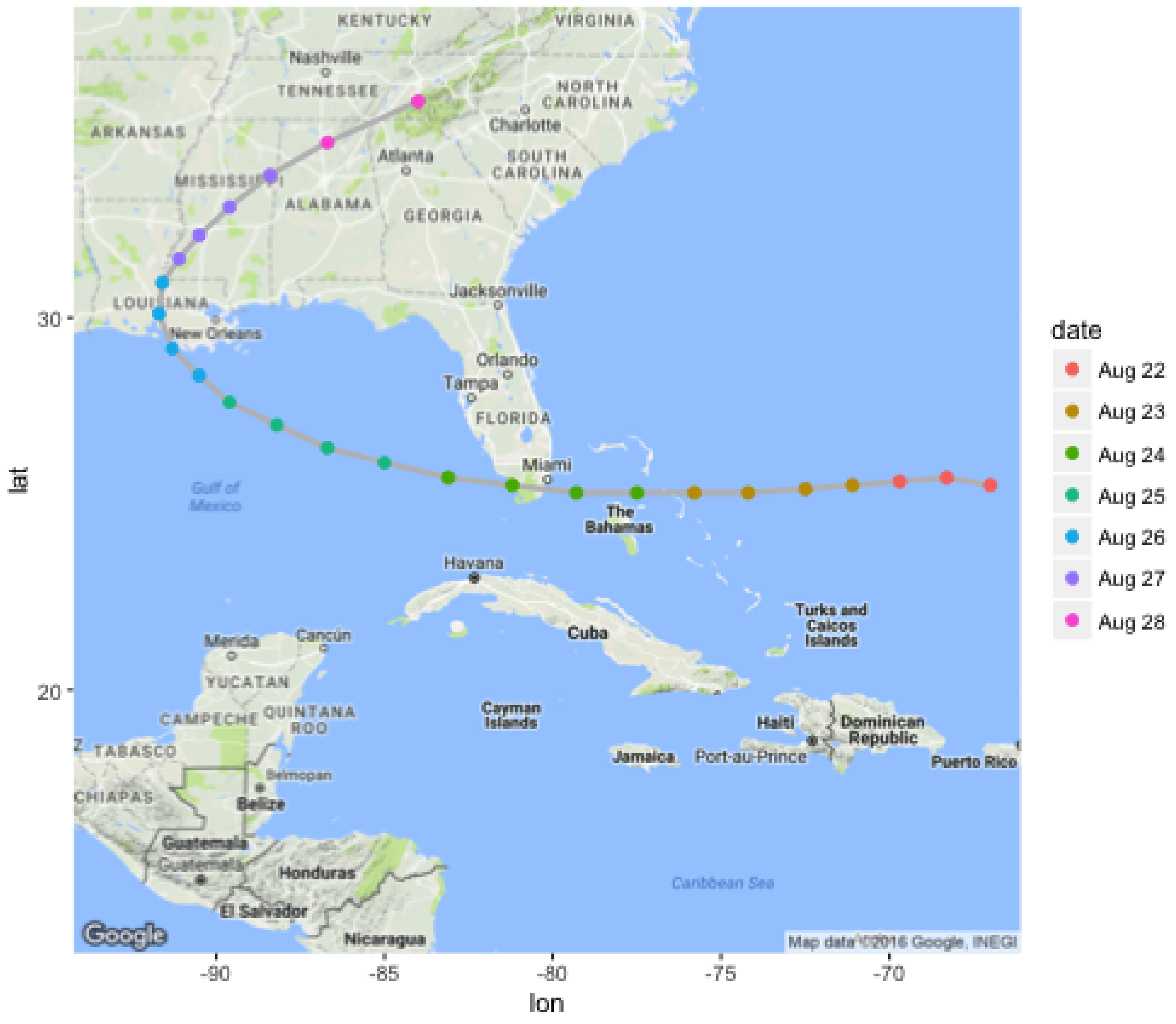
- To get the weekday and month values in the right order, the code uses the `factor` function in conjunction with the `levels` option, to control the order in which R sets the factor levels. By specifying the order we want to use with `levels`, the plot prints out using this order, rather than alphabetical order (try the code without the `factor` calls for month and weekday and compare the resulting graphs to the ones shown here).

- The `grid.arrange` function, from the `gridExtra` package, allows you to arrange different `ggplot` objects in the same plot area. Here, I've used it to put the bar charts for weekday (`a`) and for month (`b`) together in one column (`ncol = 1`).
- If you ever have `ggplot` code that you would like to re-use for a new plot with a different data frame, you can save a lot of copying and pasting by using the `%+%` function. This function takes a ggplot object (`a` in this case, which is the bar chart by weekday) and substitutes a different data frame (`check_months`) for the original one (`check_weekdays`), but otherwise maintains all code. Note that we used `rename` to give the x-variable the same name in both datasets so we could take advantage of the `%+%` function.

Working with time zones

The `lubridate` package also has functions for handling time zones. The hurricane tracks date-times are, as is true for a lot of weather data, in Coordinated Universal Time (UTC). This means that you can plot the storm track by date, but the dates will be based on UTC rather than local time near where the storm hit. Figure @ref(fig:andrewutc) shows the location of Hurricane Andrew by date as it neared and crossed the United States, based on date-time observations in UTC.

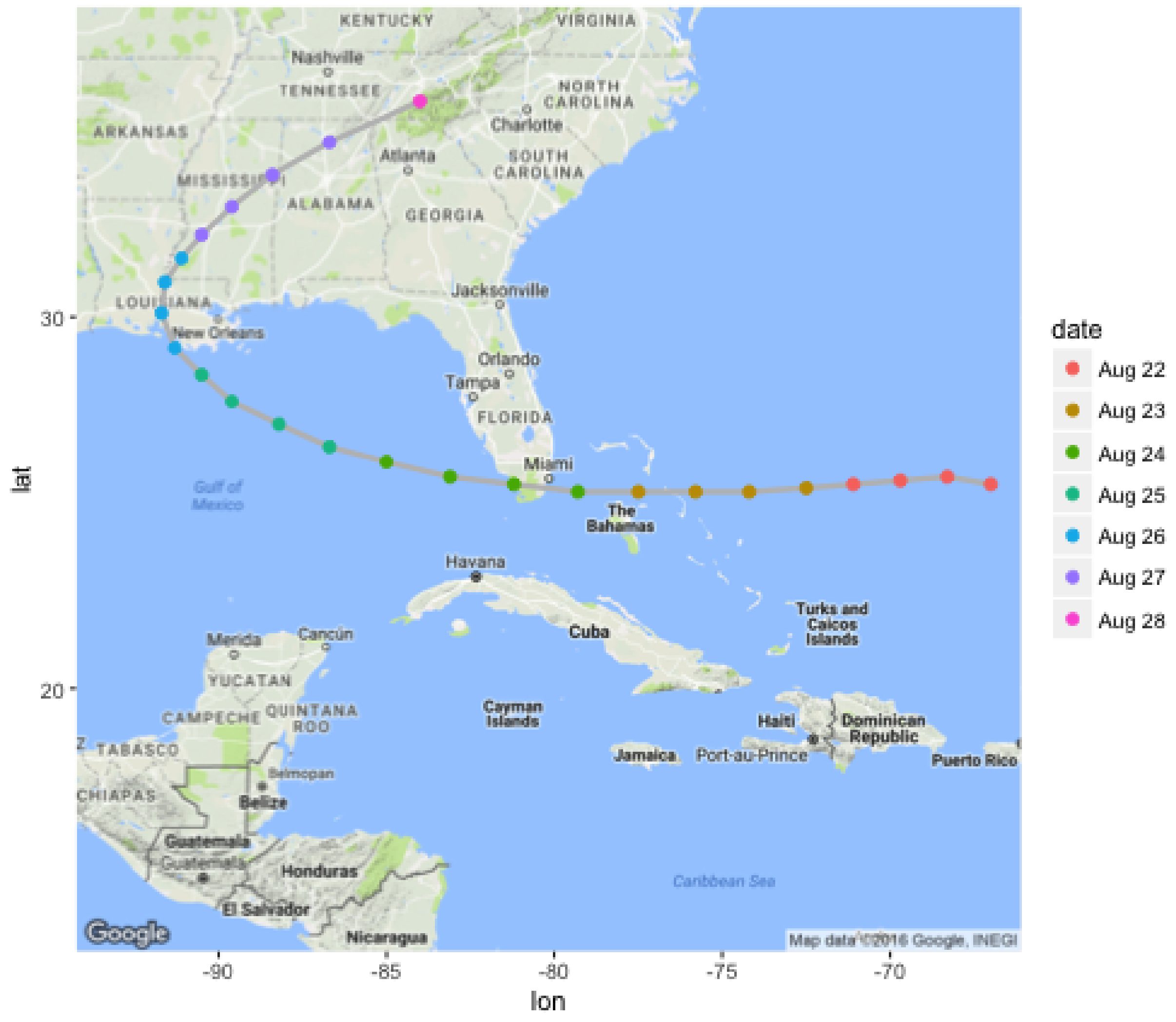
```
andrew_tracks <- ext_tracks %>%
  filter(storm_name == "ANDREW") %>%
  slice(23:47) %>%
  select(year, month, day, hour, latitude, longitude) %>%
  unite(datetime, year, month, day, hour) %>%
  mutate(datetime = ymd_h(datetime),
         date = format(datetime, "%b %d"))

library(ggmap)
miami <- get_map("miami", zoom = 5)
ggmap(miami) +
  geom_path(data = andrew_tracks, aes(x = -longitude, y = latitude),
            color = "gray", size = 1.1) +
  geom_point(data = andrew_tracks,
             aes(x = -longitude, y = latitude, color = date),
             size = 2)
```



Hurricane Andrew tracks by date, based on UTC date times.

To create this plot using local time for Miami, FL, rather than UTC (Figure @ref(fig:andrewlocal)), you can use the `with_tz` function from `lubridate` to convert the `datetime` variable in the track data from UTC to local time. This function inputs a date-time object in the `POSIXct` class, as well as a character string with the time zone of the location for which you'd like to get local time, and returns the corresponding local time for that location.

```
andrew_tracks <- andrew_tracks %>%
  mutate(datetime = with_tz(datetime, tzone = "America/New_York"),
         date = format(datetime, "%b %d"))

ggmap(miami) +
  geom_path(data = andrew_tracks, aes(x = -longitude, y = latitude),
            color = "gray", size = 1.1) +
  geom_point(data = andrew_tracks,
             aes(x = -longitude, y = latitude, color = date),
             size = 2)
```



Hurricane Andrew tracks by date, based on Miami, FL, local time.

With Figure @ref(fig:andrewlocal), it is clearer that Andrew made landfall in Florida on the morning of August 24 local time.

This section has only skimmed the surface of the date-time manipulations you can do with the `lubridate` package. For more on what this package can do, check out Garrett Grolemund and Hadley Wickham's article in the *Journal of Statistical Software* on the package– "Dates and Times Made Easy with `lubridate`"– or the current package vignette.

1.7 Text Processing and Regular Expressions

The learning objectives for this section are to:

- Transform non-tidy data into tidy data
- Manipulate and transform a variety of data types, including dates, times, and text data

Most common types of data are encoded in text, even if that text is representing numerical values, so being able to manipulate text as a software developer is essential. R provides several built-in tools for manipulating text, and there is a rich ecosystem of packages for R for text based analysis. First let's concentrate on some basic text manipulation functions.

Text Manipulation Functions in R

Text in R is represented as a string object, which looks like a phrase surrounded by quotation marks in the R console. For example `"Hello!"` and `'Strings are fun!'` are both strings. You can tell whether an object is a string using the `is.character()` function. Strings are also known as characters in R.

You can combine several strings using the `paste()` function:

```
paste("Square", "Circle", "Triangle")
[1] "Square Circle Triangle"
```

By default the `paste()` function inserts a space between each word. You can insert a different string between each word by specifying the `sep` argument:

```
paste("Square", "Circle", "Triangle", sep = "+")
[1] "Square+Circle+Triangle"
```

A shortcut for combining all of the string arguments without any characters in between each of them is to use the `paste0()` function:

```
paste0("Square", "Circle", "Triangle")
[1] "SquareCircleTriangle"
```

You can also provide a vector of strings as an argument to `paste()`. For example:

```
shapes <- c("Square", "Circle", "Triangle")
paste("My favorite shape is a", shapes)
[1] "My favorite shape is a Square"   "My favorite shape is a Circle"
[3] "My favorite shape is a Triangle"

two_cities <- c("best", "worst")
paste("It was the", two_cities, "of times.")
[1] "It was the best of times."  "It was the worst of times."
```

As you can see, all of the possible string combinations are produced when you provide a vector of strings as an argument to `paste()`. You can also collapse all of the elements of a vector of strings into a single string by specifying the `collapse` argument:

```
paste(shapes, collapse = " ")
[1] "Square Circle Triangle"
```

Besides pasting strings together, there are a few other basic string manipulation functions you should be made aware of. The `nchar()` function counts the number of characters in a string:

```
nchar("Supercalifragilisticexpialidocious")
[1] 34
```

The `toupper()` and `tolower()` functions make strings all uppercase or lowercase respectively:

```
cases <- c("CAPS", "low", "Title")
tolower(cases)
[1] "caps"  "low"   "title"
toupper(cases)
[1] "CAPS"  "LOW"   "TITLE"
```

Regular Expressions

Now that we've covered the basics of string manipulation in R, let's discuss the more advanced topic of regular expressions. A regular expression is a string that defines a pattern that could be contained within another string. A regular expression can be used for searching for a string, searching within a string, or replacing one part of a string with another string. In this section I might refer to a regular expression as a regex, just know that they're the same thing.

Regular expressions use characters to define patterns of other characters. Although that approach may seem problematic at first, we'll discuss meta-characters (characters that describe other characters) and how you can use them to create powerful regular expressions.

One of the most basic functions in R that uses regular expressions is the `grepl()` function, which takes two arguments: a regular expression and a string to be searched. If the string contains the specified regular expression then `grepl()` will return `TRUE`, otherwise it will return `FALSE`. Let's take a look at one example:

```
regular_expression <- "a"
string_to_search <- "Maryland"

grepl(regular_expression, string_to_search)
[1] TRUE
```

In the example above we specify the regular expression `"a"` and store it in a variable called `regular_expression`. Remember that regular expressions are just strings! We also store the string `"Maryland"` in a variable called `string_to_search`. The regular expression `"a"` represents a single occurrence of the character `"a"`. Since `"a"` is contained within `"Maryland"`, `grepl()` returns the value `TRUE`. Let's try another simple example:

```
regular_expression <- "u"
string_to_search <- "Maryland"

grepl(regular_expression, string_to_search)
[1] FALSE
```

The regular expression `"u"` represents a single occurrence of the character `"u"`, which is not a sub-string of `"Maryland"`, therefore `grepl()` returns the value `FALSE`. Regular expressions can be much longer than single characters. You could for example search for smaller strings inside of a larger string:

```
grepl("land", "Maryland")
[1] TRUE
grepl("ryla", "Maryland")
[1] TRUE
grepl("Marly", "Maryland")
[1] FALSE
grepl("dany", "Maryland")
[1] FALSE
```

Since `"land"` and `"ryla"` are sub-strings of `"Maryland"`, `grepl()` returns `TRUE`, however when a regular expression like `"Marly"` or `"dany"` is searched `grepl()` returns `FALSE` because neither are sub-strings of `"Maryland"`.

There's a dataset that comes with R called `state.name` which is a vector of Strings, one for each state in the United States of America. We're going to use this vector in several of the following examples.

```
head(state.name)
[1] "Alabama"    "Alaska"     "Arizona"    "Arkansas"   "California"
[6] "Colorado"
```

Let's build a regular expression for identifying several strings in this vector, specifically a regular expression that will match names of states that both start and end with a vowel.

The state name could start and end with any vowel, so we won't be able to match exact sub-strings like in the previous examples. Thankfully we can use metacharacters to look for vowels and other parts of strings. The first metacharacter that we'll discuss is `"."`. The metacharacter that only consists of a period represents any character other than a new line (we'll discuss new lines soon). Let's take a look at some examples using the peroid regex:

```
grepl(".", "Maryland")
[1] TRUE
grepl(".", "*&2[0+,%<@#~|}")
[1] TRUE
grepl(".", "")
[1] FALSE
```

As you can see the period metacharacter is very liberal. This metacharacter is most userful when you don't care about a set of characters in a regular expression. For example:

```
grepl("a.b", c("aaa", "aab", "abb", "acadb"))
[1] FALSE  TRUE  TRUE  TRUE
```

In the case above `grepl()` returns `TRUE` for all strings that contain an `a` followed by any other character followed by a `b`.

You can specify a regular expression that contains a certain number of characters or metacharacters using the enumeration metacharacters. The `+` metacharacter indicates that one or more of the preceding expression should b present and `*` indicates that zero or more of the preceding expression is present. Let's take a look at some examples using these metacharacters:

```
# Does "Maryland" contain one or more of "a" ?
grepl("a+", "Maryland")
[1] TRUE

# Does "Maryland" contain one or more of "x" ?
grepl("x+", "Maryland")
[1] FALSE

# Does "Maryland" contain zero or more of "x" ?
grepl("x*", "Maryland")
[1] TRUE
```

You can also specify exact numbers of expressions using curly brackets `{}`. For example `"a{5}"` specifies "a exactly five times," `"a{2,5}"` specifies "a between 2 and 5 times," and `"a{2,}"` specifies "a at least 2 times." Let's take a look at some examples:

```
# Does "Mississippi" contain exactly 2 adjacent "s" ?
grepl("s{2}", "Mississippi")
[1] TRUE

# This is equivalent to the expression above:
grepl("ss", "Mississippi")
[1] TRUE

# Does "Mississippi" contain between 1 and 3 adjacent "s" ?
grepl("s{2,3}", "Mississippi")
[1] TRUE

# Does "Mississippi" contain between 2 and 3 adjacent "i" ?
grepl("i{2,3}", "Mississippi")
[1] FALSE

# Does "Mississippi" contain between 2 adjacent "iss" ?
grepl("(iss){2}", "Mississippi")
[1] TRUE

# Does "Mississippi" contain between 2 adjacent "ss" ?
grepl("(ss){2}", "Mississippi")
[1] FALSE

# Does "Mississippi" contain the pattern of an "i" followed by
# 2 of any character, with that pattern repeated three times adjacently?
grepl("(i.{2}){3}", "Mississippi")
[1] TRUE
```

In the last three examples I used parentheses `()` to create a capturing group. A capturing group allows you to use quantifiers on other regular expressions. In the last example I first created the regex `"i.{2}"` which matches `i` followed by any two characters ("iss" or "ipp"). I then used a capture group to to wrap that regex, and to specify exactly three adjacent occurrences of that regex.

You can specify sets of characters with regular expressions, some of which come built in, but you can build your own character sets too. First we'll discuss the built in character sets: words (`"\\w"`), digits (`"\\d"`), and whitespace characters (`"\\s"`). Words specify any letter, digit, or a underscore, digits specify the digits 0 through 9, and whitespace specifies line breaks, tabs, or spaces. Each of these character sets have their own compliments: not words (`"\\W"`), not digits (`"\\D"`), and not whitespace characters (`"\\S"`). Each specifies all of the characters not included in their corresponding character sets. Let's take a look at a few exmaples:

```
grepl("\\w", "abcdefghijklmnopqrstuvwxyz0123456789")
[1] TRUE

grepl("\\d", "0123456789")
[1] TRUE

# "\n" this regex for a new line and "\t" is the regex for a tab
grepl("\\s", "\n\t   ")
[1] TRUE

grepl("\\d", "abcdefghijklmnopqrstuvwxyz")
[1] FALSE

grepl("\\D", "abcdefghijklmnopqrstuvwxyz")
[1] TRUE

grepl("\\w", "\n\t   ")
[1] FALSE
```

You can also specify specific character sets using straight brackets `[]`. For example a character set of just the vowels would look like: `"[aeiou]"`. You can find the complement to a specific character by putting a carrot `^` after the first bracket. For example `"[^aeiou]"` matches all characters except the lowercase vowels. You can also specify ranges of characters using a hyphen `-` inside of the brackets. For example `"[a-m]"` matches all of the lowercase characters between `a` and `m`, while `"[5-8]"` matches any digit between 5 and 8 inclusive. Let's take a look at some examples using custom character sets:

```
grepl("[aeiou]", "rhythms")
[1] FALSE

grepl("[^aeiou]", "rhythms")
[1] TRUE

grepl("[a-m]", "xyz")
[1] FALSE

grepl("[a-m]", "ABC")
[1] FALSE

grepl("[a-mA-M]", "ABC")
[1] TRUE
```

You might be wondering how you can use regular expressions to match a particular punctuation mark since many punctuation marks are used as metacharacters! Putting two backslashes before a punctuation mark that is also a metacharacter indicates that you are looking for the symbol and not the metacharacter meaning. For example `"\\."` indicates you are trying to match a period in a string. Let's take a look at a few examples:

```r
grepl("\\+", "tragedy + time = humor")
[1] TRUE

grepl("\\.", "http://www.jhsph.edu/")
[1] TRUE
```

There are also metacharacters for matching the beginning and the end of a string which are `"^"` and `"$"` respectively. Let's take a look at a few examples:

```r
grepl("^a", c("bab", "aab"))
[1] FALSE  TRUE

grepl("b$", c("bab", "aab"))
[1] TRUE TRUE

grepl("^[ab]+$", c("bab", "aab", "abc"))
[1]  TRUE  TRUE FALSE
```

The last metacharacter we'll discuss is the OR metacharacter (`"|"`). The OR metacharacter matches either the regex on the left or the regex on the right side of this character. A few examples:

```r
grepl("a|b", c("abc", "bcd", "cde"))
[1]  TRUE  TRUE FALSE

grepl("North|South", c("South Dakota", "North Carolina", "West Virginia"))
[1]  TRUE  TRUE FALSE
```

Finally we've learned enough to create a regular expression that matches all state names that both begin and end with a vowel:

1. We match the beginning of a string.
2. We create a character set of just capitalized vowels.
3. We specify one instance of that set.
4. Then any number of characters until:
5. A character set of just lowercase vowels.
6. We specify one instance of that set.
7. We match the end of a string.

```
start_end_vowel <- "^[AEIOU]{1}.+[aeiou]{1}$"
vowel_state_lgl <- grepl(start_end_vowel, state.name)
head(vowel_state_lgl)
[1]  TRUE  TRUE  TRUE FALSE FALSE FALSE

state.name[vowel_state_lgl]
[1] "Alabama"  "Alaska"   "Arizona"  "Idaho"    "Indiana"  "Iowa"
[7] "Ohio"     "Oklahoma"
```

Below is a table of several important metacharacters:

Metacharacter	Meaning
.	Any Character
\w	A Word
\W	Not a Word
\d	A Digit
\D	Not a Digit
\s	Whitespace
\S	Not Whitespace
[xyz]	A Set of Characters
[^xyz]	Negation of Set
[a-z]	A Range of Characters
^	Beginning of String
$	End of String
\n	Newline
+	One or More of Previous
*	Zero or More of Previous
?	Zero or One of Previous
	Either the Previous or the Following
{5}	Exactly 5 of Previous
{2, 5}	Between 2 and 5 or Previous
{2, }	More than 2 of Previous

RegEx Functions in R

So far we've been using `grepl()` to see if a regex matches a string. There are a few other built in reged functions you should be aware of. First we'll review our workhorse of this chapter, `grepl()` which stands for "grep logical."

```
grepl("[Ii]", c("Hawaii", "Illinois", "Kentucky"))
[1]  TRUE  TRUE FALSE
```

Then there's old fashioned `grep()` which returns the indices of the vector that match the regex:

```
grep("[Ii]", c("Hawaii", "Illinois", "Kentucky"))
[1] 1 2
```

The `sub()` function takes as arguments a regex, a "replacement," and a vector of strings. This function will replace the first instance of that regex found in each string.

```
sub("[Ii]", "1", c("Hawaii", "Illinois", "Kentucky"))
[1] "Hawa1i"   "1llinois" "Kentucky"
```

The `gsub()` function is nearly the same as `sub()` except it will replace every instance of the regex that is matched in each string.

```
gsub("[Ii]", "1", c("Hawaii", "Illinois", "Kentucky"))
[1] "Hawa11"   "1ll1no1s" "Kentucky"
```

The `strsplit()` function will split up strings according to the provided regex. If `strsplit()` is provided with a vector of strings it will return a list of string vectors.

```
two_s <- state.name[grep("ss", state.name)]
two_s
[1] "Massachusetts" "Mississippi"   "Missouri"      "Tennessee"
strsplit(two_s, "ss")
[[1]]
[1] "Ma"        "achusetts"

[[2]]
[1] "Mi"   "i"    "ippi"

[[3]]
[1] "Mi"   "ouri"

[[4]]
[1] "Tenne" "ee"
```

The stringr Package

The `stringr` package, written by Hadley Wickham, is part of the Tidyverse group of R packages. This package takes a "data first" approach to functions involving regex, so usually the string is the first argument and the regex is the second argument. The majority of the function names in `stringr` begin with `str_`.

The `str_extract()` function returns the sub-string of a string that matches the providied regular expression.

```
library(stringr)
state_tbl <- paste(state.name, state.area, state.abb)
head(state_tbl)
[1] "Alabama 51609 AL"     "Alaska 589757 AK"     "Arizona 113909 AZ"
[4] "Arkansas 53104 AR"    "California 158693 CA" "Colorado 104247 CO"
str_extract(state_tbl, "[0-9]+")
 [1] "51609"  "589757" "113909" "53104"  "158693" "104247" "5009"
 [8] "2057"   "58560"  "58876"  "6450"   "83557"  "56400"  "36291"
[15] "56290"  "82264"  "40395"  "48523"  "33215"  "10577"  "8257"
[22] "58216"  "84068"  "47716"  "69686"  "147138" "77227"  "110540"
[29] "9304"   "7836"   "121666" "49576"  "52586"  "70665"  "41222"
[36] "69919"  "96981"  "45333"  "1214"   "31055"  "77047"  "42244"
[43] "267339" "84916"  "9609"   "40815"  "68192"  "24181"  "56154"
[50] "97914"
```

The `str_order()` function returns a numeric vector that corresponds to the alphabetical order of the strings in the provided vector.

```
head(state.name)
[1] "Alabama"    "Alaska"     "Arizona"    "Arkansas"   "California"
[6] "Colorado"
str_order(state.name)
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
[24] 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
[47] 47 48 49 50

head(state.abb)
[1] "AL" "AK" "AZ" "AR" "CA" "CO"
str_order(state.abb)
 [1]  2  1  4  3  5  6  7  8  9 10 11 15 12 13 14 16 17 18 21 20 19 22 23
[24] 25 24 26 33 34 27 29 30 31 28 32 35 36 37 38 39 40 41 42 43 44 46 45
[47] 47 49 48 50
```

The `str_pad()` function pads strings with other characters which is often useful when the string is going to be eventually printed for a person to read.

```
str_pad("Thai", width = 8, side = "left", pad = "-")
[1] "----Thai"
str_pad("Thai", width = 8, side = "right", pad = "-")
[1] "Thai----"
str_pad("Thai", width = 8, side = "both", pad = "-")
[1] "--Thai--"
```

The `str_to_title()` function acts just like `tolower()` and `toupper()` except it puts strings into Title Case.

```
cases <- c("CAPS", "low", "Title")
str_to_title(cases)
[1] "Caps"  "Low"   "Title"
```

The `str_trim()` function deletes whitespace from both sides of a string.

```
to_trim <- c("   space", "the    ", "    final frontier  ")
str_trim(to_trim)
[1] "space"          "the"            "final frontier"
```

The `str_wrap()` function inserts newlines in strings so that when the string is printed each line's length is limited.

```
pasted_states <- paste(state.name[1:20], collapse = " ")

cat(str_wrap(pasted_states, width = 80))
Alabama Alaska Arizona Arkansas California Colorado Connecticut Delaware Florida
Georgia Hawaii Idaho Illinois Indiana Iowa Kansas Kentucky Louisiana Maine
Maryland
cat(str_wrap(pasted_states, width = 30))
Alabama Alaska Arizona
Arkansas California Colorado
Connecticut Delaware Florida
Georgia Hawaii Idaho Illinois
Indiana Iowa Kansas Kentucky
Louisiana Maine Maryland
```

The `word()` function allows you to index each word in a string as if it were a vector.

```
a_tale <- "It was the best of times it was the worst of times it was the age of wisdom it was the ag\
e of foolishness"

word(a_tale, 2)
[1] "was"

word(a_tale, end = 3)
[1] "It was the"

word(a_tale, start = 11, end = 15)
[1] "of times it was the"
```

Summary

String manipulation in R is useful for data cleaning, plus it can be fun! For prototyping your first regular expressions I highly recommend checking out http://regexr.com/. If you're interested in what some people call a more "humane" way of constructing regular expressions you should check out the rex package by Kevin Ushey and Jim Hester. If you'd like to find out more about text analysis I highly recommend reading Tidy Text Mining in R by Julia Silge and David Robinson.

1.8 The Role of Physical Memory

The learning objectives of this section are to:

- Describe how memory is used in R sessions to store R objects

Generally speaking, R stores and manipulates all objects in the physical memory of your computer (i.e. the RAM). Therefore, it's important to be aware of the limits of your computing environment with respect to available memory and how that may affect your ability to use R. In the event that your computer's physical memory is insufficient for some of your work, there have been some developments that allow R users to deal with objects out of physical memory and we will discuss them below.

The first thing that is worth keeping in mind as you use R is how much physical memory your computer actually has. Typically, you can figure this out by looking at your operating system's settings. For example, as of this writing, Roger has a 2015-era Macbook with 8 GB of RAM. Of course, the amount of RAM available to R will be quite a bit less than that, but it's a useful upper bound. If you plan to read into R an object that is 16 GB on this computer, you're going to have ask Roger for a new computer.

The `pryr` package provides a number of useful functions for interrogating the memory usage of your R session. Perhaps the most basic is the `mem_used()` function, which tells you how much memory your current R session is using.

```
library(pryr)
mem_used()
125 MB
```

The primary use of this function is to make sure your memory usage in R isn't getting too big. If the output from `mem_used()` is in the neighborhood of 75%-80% of your total physical RAM, you might need to consider a few things.

First, you might consider removing a few very large objects in your workspace. You can see the memory usage of objects in your workspace by calling the `object_size()` function.

```
ls()  ## Show objects in workspace
 [1] "a"                 "a_tale"             "andrew_tracks"
 [4] "b"                 "cases"              "check_months"
 [7] "check_tracks"      "check_weekdays"     "denver"
[10] "ext_tracks"        "ext_tracks_colnames" "ext_tracks_file"
[13] "ext_tracks_widths" "input"              "join_funcs"
[16] "katrina"           "katrina_reduced"    "knots_to_mph"
[19] "logdates"          "logs"               "m"
[22] "mc_tibl"           "meso_url"           "miami"
[25] "msg"               "old"                "pasted_states"
[28] "readr_functions"   "regular_expression" "shapes"
[31] "start_end_vowel"   "state_tbl"          "string_to_search"
```

```
[34] "team_standings"      "teams"               "to_trim"
[37] "two_cities"          "two_s"               "VADeaths"
[40] "vowel_state_lgl"     "wc_table"            "worldcup"
[43] "x"                   "y"                   "zika_brazil"
[46] "zika_file"
object_size(worldcup)
61.2 kB
```

The `object_size()` function will print the number of bytes (or kilobytes, or megabytes) that a given object is using in your R session. If you want see what the memory usage of the largest 5 objects in your workspace is, you can use the following code.

```
library(magrittr)
sapply(ls(), function(x) object_size(get(x))) %>% sort %>% tail(5)
  ext_tracks        miami            a            b knots_to_mph
     1834904     13121552     15713360     15713360     15713360
```

Here we can see that the `miami` and `ext_tracks` objects (created in previous chapters of this book) are currently taking up the most memory in our R session. Since we no longer need those objects, we can remove them from the workspace and free up some memory.

```
mem_used()
125 MB
rm(ext_tracks, miami)
mem_used()
123 MB
```

Here you can see how much memory we save by deleting these two objects. But you may be wondering why there isn't a larger savings, given the number reported by `object_size()`. This has to do with the internal representation of the `miami` object, which is of the class `ggmap`. Occasionally, certain types of R objects can appear to take up more memory than the actually do, in which case functions like `object_size()` will get confused.

Viewing the change in memory usage by executing an R expression can actually be simplified using the `mem_change()` function. We can see what happens when we remove the next three largest objects.

```
mem_change(rm(check_tracks, denver, b))
-458 kB
```

Here the decrease is about 450 KB.

R has a built in function called `object.size()` that also calculates the size of an object, but it uses a slightly different calculation than `object_size()` in `pryr`. While the two functions will generally agree for most objects, for things like functions and formulas, which have enclosing environments attached to them, they will differ. Similarly, objects with shared elements (i.e. character vectors) may result in different computations of their size. The `compare_size()` function in `pryr` allows you to see how the two functions compare in their calculations. We will discuss these concepts more in the next chapter.

Back of the Envelope Calculations

When reading in large datasets or creating large R objects, it's often useful to do a back of the envelope calculation of how much memory the object will occupy in the R session (ideally *before* creating the object). To do this it's useful to know roughly how much memory different types of atomic data types in R use.

It's difficult to generalize how much memory is used by data types in R, but on most 64 bit systems today, integers are 32 bits (4 bytes) and double-precision floating point numbers (numerics in R) are 64 bits (8 bytes). Furthermore, character data are usually 1 byte per character. Because most data come in the form of numbers (integer or numeric) and letters, just knowing these three bits of information can be useful for doing many back of the envelope calculations.

For example, an integer vector is roughly 4 bytes times the number of elements in the vector. We can see that for a zero-length vector, that still requires some memory to represent the data structure.

```
object_size(integer(0))
40 B
```

However, for longer vectors, the overhead stays roughly constant, and the size of the object is determined by the number of elements.

```
object_size(integer(1000))  ## 4 bytes per integer
4.04 kB
object_size(numeric(1000))  ## 8 bytes per numeric
8.04 kB
```

If you are reading in tabular data of integers and floating point numbers, you can roughly estimate the memory requirements for that table by multiplying the number of rows by the memory required for each of the columns. This can be a useful exercise to do before reading in large datasets. If you accidentally read in a dataset that requires more memory than your computer has available, you may end up freezing your R session (or even your computer).

The `.Machine` object in R (found in the `base` package) can give you specific details about how your computer/operation system stores different types of data.

```
str(.Machine)
List of 18
 $ double.eps           : num 2.22e-16
 $ double.neg.eps       : num 1.11e-16
 $ double.xmin          : num 2.23e-308
 $ double.xmax          : num 1.8e+308
 $ double.base          : int 2
 $ double.digits        : int 53
 $ double.rounding      : int 5
 $ double.guard         : int 0
 $ double.ulp.digits    : int -52
 $ double.neg.ulp.digits: int -53
 $ double.exponent      : int 11
 $ double.min.exp       : int -1022
 $ double.max.exp       : int 1024
 $ integer.max          : int 2147483647
 $ sizeof.long          : int 8
 $ sizeof.longlong      : int 8
 $ sizeof.longdouble    : int 16
 $ sizeof.pointer       : int 8
```

The floating point representation of a decimal number contains a set of bits representing the *exponent* and another set of bits representing the *significand* or the *mantissa*. Here the number of bits used for the exponent is 11, from `double.exponent`, and the number of bits for the significand is 53, from the `double.digits` element. Together, each double precision floating point number requires 64 bits, or 8 bytes to store.

For integers, we can see that the maximum integer indicated by the `integer.max` is 2147483647, we can take the base 2 log of that number and see that it requires 31 bits to encode. Because we need another bit to encode the sign of the number, the total number of bits for an integer is 32, or 4 bytes.

Much of the point of this discussion of memory is to determine if your computer has sufficient memory to do the work you want to do. If you determine that the data you're working with cannot be completely stored in memory for a given R session, then you may need to resort to alternate tactics. We discuss one such alternative in the section below, "Working with large datasets".

Internal Memory Management in R

If you're familiar with other programming languages like C, you'll notice that you do not need to explicitly allocate and de-allocate memory for objects in R. This is because R has a garbage collection system that recycles unused memory and gives it back to R. This happens automatically without the need for user intervention.

Roughly, R will periodically cycle through all of the objects that have been created and see if there are still any references to the object somewhere in the session. If there are no references, the object is garbage-collected and the memory returned. Under normal usage, the garbage collection is not noticeable, but occasionally, when working with very large R objects, you may notice a "hiccup" in your R session when R triggers a garbage collection to reclaim unused memory. There's not really anything you can do about this except not panic when it happens.

The `gc()` function in the `base` package can be used to explicitly trigger a garbage collection in R. Calling `gc()` explicitly is never actually needed, but it does produce some output that is worth understanding.

```
gc()
          used (Mb) gc trigger  (Mb) max used  (Mb)
Ncells 1659083 88.7    2637877 140.9  2637877 140.9
Vcells 3761082 28.7   11511995  87.9 18885262 144.1
```

The `used` column gives you the amount of memory currently being used by R. The distinction between `Ncells` and `Vcells` is not important—the `mem_used()` function in `pryr` essentially gives you the sum of this column. The `gc trigger` column gives you the amount of memory that can be used before a garbage collection is triggered. Generally, you will see this number go up as you allocate more objects and use more memory. The `max used` column shows the maximum space used since the last call to `gc(reset = TRUE)` and is not particularly useful.

1.9 Working with Large Datasets

The learning objectives of this section are to:

- Read and manipulate large datasets

R now offers now offers a variety of options for working with large datasets. We won't try to cover all these options in detail here, but rather give an overview of strategies to consider if you need to work with a large dataset, as well as point you to additional resources to learn more about working with large datasets in R.

> While there are a variety of definitions of how large a dataset must be to qualify as "large", in this section we don't formally define a limit. Instead, this section is meant to give you some strategies anytime you work with a dataset large enough that you notice it's causing problems. For example, data large enough for R to be noticeably slow to read or manipulate the data, or large enough it's difficult to store the data locally on your computer.

In-memory strategies

In this section, we introduce the basics of why and how to use `data.table` to work with large datasets in R. We have included a video demonstration online showing how functions from the `data.table` package can be used to load and explore a large dataset more efficiently.

The `data.table` package can help you read a large dataset into R and explore it more efficiently. The `fread` function in this package, for example, can read large flat files in much more quickly than comparable base R packages. Since all of the `data.table` functions will work with smaller datasets, as well, we'll illustrate using `data.table` with the Zika data accessed from GitHub in an earlier section of this chapter. We've saved that data locally to illustrate how to read it in and work with it using `data.table`.

First, to read this data in using `fread`, you can run:

```
library(data.table)
brazil_zika <- fread("data/COES_Microcephaly-2016-06-25.csv")
head(brazil_zika, 2)
   report_date       location location_type             data_field
1:  2016-06-25    Brazil-Acre         state microcephaly_confirmed
2:  2016-06-25 Brazil-Alagoas         state microcephaly_confirmed
   data_field_code time_period time_period_type value  unit
1:          BR0002          NA               NA     2 cases
2:          BR0002          NA               NA    75 cases
class(brazil_zika)
[1] "data.table" "data.frame"
```

If you are working with a very large dataset, `data.table` will provide a status bar showing your progress towards loading the code as you read it in using `fread`.

If you have a large dataset for which you only want to read in certain columns, you can save time when using `data.table` by only reading in the columns you want with the `select` argument in `fread`. This argument takes a vector of either the names or positions of the columns that you want to read in:

```
fread("data/COES_Microcephaly-2016-06-25.csv",
      select = c("location", "value", "unit")) %>%
  dplyr::slice(1:3)
        location value  unit
1    Brazil-Acre     2 cases
2 Brazil-Alagoas    75 cases
3   Brazil-Amapa     7 cases
```

Many of the `fread` arguments are counterparts to arguments in the `read.table` family of functions in base R (for example, `na.strings`, `sep`, `skip`, `colClasses`). One that is particular useful is `nrows`. If you're working with data that takes a while to read in, using `nrows = 20` or some other small number will allow you to make sure you have set all of the arguments in `fread` appropriately for the dataset before you read in the full dataset.

If you already have a dataset loaded to your R session, you can use the `data.table` function to convert a data frame into a `data.table` object. (Note: if you use `fread`, the data is automatically read into a `data.table` object.) A `data.table` object also has the class `data.frame`; this means that you can use all of your usual methods for manipulating a data frame with a `data.table` object. However, for extra speed, use `data.table` functions to manipulate, clean, and explore the data in a `data.table` object. You can find out more about using `data.table` functions at the `data.table` wiki.

> Many of the functions in `data.table`, like many in `ddplyr`, use non-standard evaluation. This means that, while they'll work fine in interactive programming, you'll need to take some extra steps when you use them to write functions for packages. We'll cover non-standard evaluation in the context of developing packages in a later section.

When you are working with datasets that are large, but can still fit in-memory, you'll want to optimize your code as much as possible. There are more details on profiling and optimizing code in a later chapter, but one strategy for speeding up R code is to write some of the code in C++ and connect it to R using the `Rcpp` package. Since C++ is a compiled rather than an interpreted language, it runs much faster than similar code written in R. If you are more comfortable coding in another compiled language (C or FORTRAN, for example), you can also use those, although the `Rcpp` package is very nicely written and well-maintained, which makes C++ an excellent first choice for creating compiled code to speed up R.

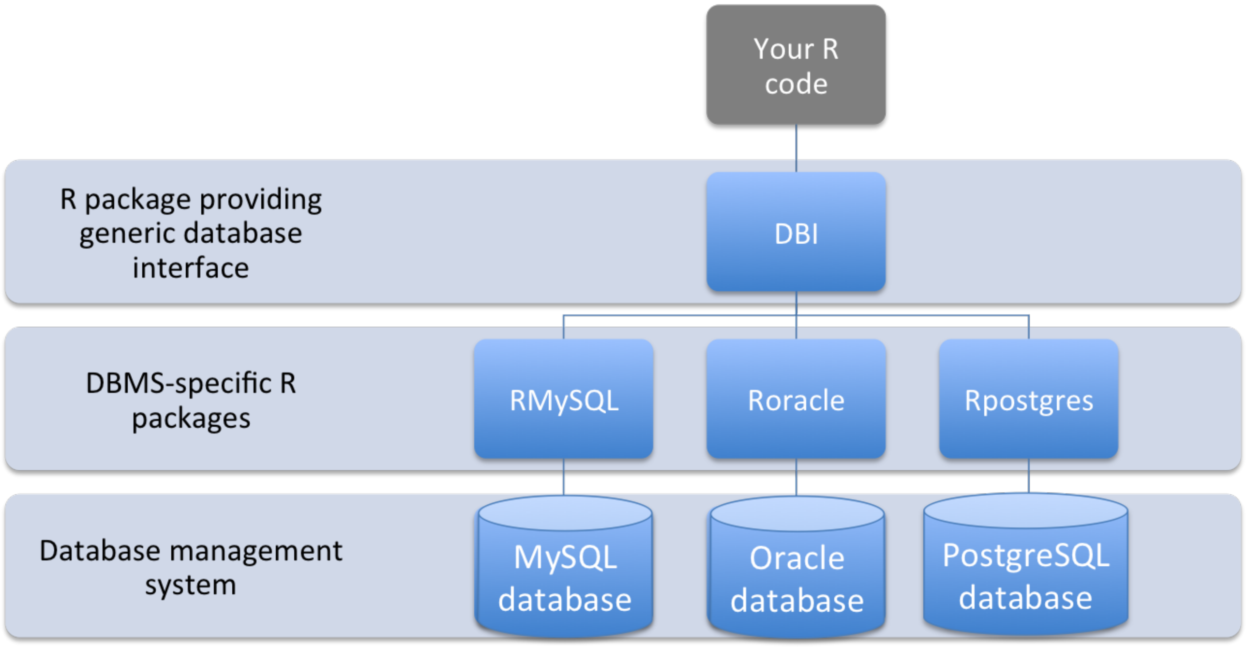
Further, a variety of R packages have been written that help you run R code in parallel, either locally or on a cluster. Parallel strategies may be work pursuing if you are working with very large datasets, and if the coding tasks can be split to run in parallel. To get more ideas and find relevant packages, visit CRAN's High-Performance and Parallel Computing with R task view.

Out-of-memory strategies

If you need to work with a very large dataset, there are also some options to explore and model the dataset without ever loading it into R, while still using R commands and working from the R console or an R script. These options can make working with large datasets more efficient, because they let other software handle the heavy lifting of sifting through the data and / or avoid loading large datasets into RAM, instead using data stored on hard drive.

For example, database management systems are optimized to more efficiently store and better search through large sets of data; popular examples include Oracle, MySQL, and PostgreSQL. There are several R packages that allow you to connect your R session to a database. With these packages, you can use functions from the R console or an R script to search and subset data without loading the whole dataset into R, and so take advantage of the improved efficiency of the database management system in handling data, as well as work with data too big to fit in memory.

The `DBI` package is particularly convenient for interfacing R code with a database management system, as it provides a top-level interface to a number of different database management systems, with system-specific code applied by a lower-level, more specific R package (Figure @ref(fig:rdbi)).



Structure of interface between code in an R script and data stored in a database management system using DBI-compliant packages

The `DBI` package therefore allows you to use the same commands for working with database-stored data in R, without worrying about details specific to the exact type of database management system you're connecting to. The following table outlines the `DBI` functions you can use to perform a variety of tasks when working with data stored in a database:

Task	DBI Function
Create a new driver object for an instance of a database	`dbDriver`
Connect to database instance	`dbConnect`
Find available tables in a connected database instance	`dbListTables`
Find available fields within a table	`dbListFields`
Query a connected database instance	`dbSendQuery`
Pull a data frame into R from a query result	`dbFetch`
Jointly query and pull data from a database instance	`dbGetQuery`
Close result set from a query	`dbClearResult`
Write a new table in a database instance	`dbWriteTable`
Remove a table from a database instance	`dbRemoveTable`
Disconnect from a database instance	`dbDisconnect`

The `DBI` package depends on lower-level R packages to translate its generic commands to work for specific database management systems. DBI-compliant R packages have not been written for every database management system, so there are some databases for which DBI commands will not work. DBI-compliant R packages that are available include:

Database Management System	**R packages**
Oracle	`ROracle`
MySQL	`RMySQL`
Microsoft SQL Server	`RSQLServer`
PostgreSQL	`RPostgres`
SQLite	`RSQLite`

For more on the `DBI` package, including its history, see the package's GitHub README page.

The packages for working with database management systems require you to send commands to the database management system in that system's command syntax (e.g., SQL). You can, however, do "SELECT" database queries directly using `dplyr` syntax for some database systems, rather than with SQL syntax. While this functionality is limited to "SELECT" calls, often this is all you'll need within a data analysis script. For more details, see the dplyr database vignette.

In addition to database management systems, there are other options for working with large data out-of-memory in R. For example, the `bigmemory` and associated packages can be used to access and work with large matrices stored on hard drive rather than in RAM, by storing the data in a C++ matrix structure and loading to R pointers to the data, rather than the full dataset. This family of packages includes packages that can be used to summarize and model the data (`biglm`, `bigglm`, `biganalytics`, `bigtabulate`, `bigalgebra`). One limitation is that these packages only work with matrices, not data frames; matrices require all elements share a class (e.g., all numeric).

Finally, there are some packages that allow you to write R code that uses other software to load and work with data through an R API provided by the other software. For example, the `h2o` package allows you to write R code to load and fit machine learning models in H2O, which is open-source software that facilitates distributed machine learning. H2O includes functions to fit and evaluate numerous machine learning models, including ensemble models, which would take quite a while to fit within R with a large training dataset. Since processing is done using compiled code, models can be fit on large datasets more quickly. However, while the `h2o` package allows you to use R-like code from within an R console to explore and model your data, it is not actually running R, but instead is using the R code, through the R API, to run Java-encoded functions. As a result, you only have access to a small subset of R's total functionality, since you can only run the R-like functions written into H2O's own software.

1.10 Diagnosing Problems

The learning objectives of this section are to:

- Describe how to diagnose programming problems and to look up answers from the web or forums

Inevitably, no matter what your level of expertise, you will get to a point in your R programming where you're stuck. It happens to us every single day.

The first question is always "How do you know you have a problem?" Two things must be satisfied in this situation:

1. You had a certain expectation for what was supposed to happen
2. Something *other* than that expectation actually happened

While it might seem overly didactic to separate out these two things, one common mistake is to only focus on the second part, i.e. what actually happened. Typically, we see an error message or a warning or some other bad sign and we intuitively know that there is a problem. While it's important to recognize these warning signs, it's equally important to be able to say specifically what your expectation was. What output were you expecting to see? What did you think the answer was going to be?

The more specific you can be with your expectation, the more likely you'll be able to figure out what went wrong. In particular, in many cases it might be that your expectations were incorrect. For example, you might think it's a bug that the `log()` function returns `NaN` when called on a negative number. If you were expecting there to be an error in this situation, then your expectation is incorrect because the `log()` function was specifically designed to return the `NaN` value (indicating an undefined operation) and give a warning when called with negative numbers.

There are two basic approaches to diagnosing and solving problems.

1. Googling
2. Asking a human

Before asking a human, it's usually best to see if you can Google your way out. This can be a real timesaver for all involved. We discuss both approaches below.

How to Google Your Way Out of a Jam

Like with any other programming language, it's essential that you know how to Google your way out of a jam. A related resource in this situation is the Stack Overflow web site, which is a popular Q&A web site for programming related questions. However, often results from Google will simply point you to Stack Overflow, so Google can serve as useful wrapper around a variety of web site like this.

While we don't exactly have an algorithm for getting unstuck from a jam, here are few tips.

- If you get an error message, **copy and paste the entire error message into Google**. Why? Because, almost surely, someone else has gotten this very same error and has asked a question about it on some forum that Google has indexed. Chances are, that person copy-and-pasted the error message into that forum posting and, presto! You have your answer. Or something close to it.
- For working with certain high-level functions, you can simply Google the name of the function, perhaps with the phrase "R function" following it in case it is a somewhat generic function name. This will usually bring up the help page for the function first, but it will also commonly bring up various tutorials that people have written that use this function. Often, seeing how other people use a certain function can be very helpful in understanding how a function works.
- If you're trying to learn a new R package, Google "[package name] vignette" and "[package name] tutorial". Often, someone will have written course slides, a blog post, or a document that walks you through how to use the package.
- If you are struggling with how to write the code for a plot, try using Google Images. Google "r [name or description of plot]" (e.g., "r pareto plot") and then choose the "Images" tab in the results. Scroll through to find something that looks like the plot you want to create, and then check the image's website. It will often include the R code used to create the image.

Asking for Help

In the event that Googling around does not find you an answer, you may need to wade into a forum like Stack Overflow, Reddit, or perhaps the R-help mailing list to get help with a problem. When asking questions on a forum, there are some general rules that are always worth following.

- Read the posting guide for the forum, if there is one. This may cover the rules of posting to the forum and will save you a bit of grief later on.
- If the forum has a FAQ, read it. The answer to your question may already be there.
- State the problem you're trying to solve, along with the approach that you took that lead to your problem. In particular, **state what you were expecting to see from your code**. Sometimes the source of your problem lies higher up the chain than you might think. In particular, **your expectations may be incorrect**.
- Show that you've done your homework and have tried to diagnose the problem yourself, read the help page, Googled for answers, etc.
- **Provide a reproducible example of your problem**. This cannot be stressed enough. In order for others to help you, it's critical that they can reproduce the problem on their own machines. Otherwise, they will have to diagnose your problem from afar, and much like with human beings, this is often very difficult to do. If your problem involves massive amounts of computation, try to come up with a simple example that reproduces the same problem. Other people will not download your 100 GB dataset just so they can reproduce your error message.

2. Advanced R Programming

This course covers advanced topics in R programming that are necessary for developing powerful, robust, and reusable data science tools. Topics covered include functional programming in R, robust error handling, object oriented programming, profiling and benchmarking, debugging, and proper design of functions. Upon completing this course you will be able to identify and abstract common data analysis tasks and to encapsulate them in user-facing functions. Because every data science environment encounters unique data challenges, there is always a need to develop custom software specific to your organization's mission. You will also be able to define new data types in R and to develop a universe of functionality specific to those data types to enable cleaner execution of data science tasks and stronger reusability within a team.

The learning objectives of the chapter are:

- Describe the control flow of an R program
- Write a function that abstracts a single concept/procedure
- Describe functional programming concepts
- Write functional programming code using the `purrr` package
- Manipulate R expressions to "compute on the language"
- Describe the semantics of R environments
- Implement exception handling routines in R functions
- Design and Implement a new S3, S4, or reference class with generics and methods
- Apply debugging tools to identify bugs in R programs
- Apply profiling and timing tools to optimize R code
- Describe the principles of tidyverse functions

2.1 Control Structures

Note: Some of the material in this section is adapted from R Programming for Data Science.

The learning objectives of the section are:

- Describe the control flow of an R program

Control structures in R allow you to control the flow of execution of a series of R expressions. Basically, control structures allow you to put some "logic" into your R code, rather than just always executing the same R code every time. Control structures allow you to respond to inputs or to features of the data and execute different R expressions accordingly.

Commonly used control structures are

- `if` and `else`: testing a condition and acting on it
- `for`: execute a loop a fixed number of times
- `break`: break the execution of a loop
- `next`: skip an iteration of a loop

Most control structures are not used in interactive sessions, but rather when writing functions or longer expressions. However, these constructs do not have to be used in functions and it's a good idea to become familiar with them before we delve into functions.

`if-else`

The `if-else` combination is probably the most commonly used control structure in R (or perhaps any language). This structure allows you to test a condition and act on it depending on whether it's true or false.

For starters, you can just use the `if` statement.

```
if(<condition>) {
        ## do something
}
## Continue with rest of code
```

The above code does nothing if the condition is false. If you have an action you want to execute when the condition is false, then you need an `else` clause.

```
if(<condition>) {
        ## do something
} else {
        ## do something else
}
```

You can have a series of tests by following the initial `if` with any number of `else if`s.

```
if(<condition1>) {
        ## do something
} else if(<condition2>)  {
        ## do something different
} else {
        ## do something different
}
```

Here is an example of a valid if/else structure.

```
## Generate a uniform random number
x <- runif(1, 0, 10)
if(x > 3) {
        y <- 10
} else {
        y <- 0
}
```

The value of `y` is set depending on whether `x > 3` or not.

Of course, the `else` clause is not necessary. You could have a series of if clauses that always get executed if their respective conditions are true.

```
if(<condition1>) {

}

if(<condition2>) {

}
```

`for` Loops

For loops are pretty much the only looping construct that you will need in R. While you may occasionally find a need for other types of loops, in most data analysis situations, there are very few cases where a for loop isn't sufficient.

In R, for loops take an iterator variable and assign it successive values from a sequence or vector. For loops are most commonly used for iterating over the elements of an object (list, vector, etc.)

```
numbers <- rnorm(10)
for(i in 1:10) {
        print(numbers[i])
}
[1] -0.9567815
[1] 1.347491
[1] -0.03158058
[1] 0.5960358
[1] 1.133312
[1] -0.7085361
[1] 1.525453
[1] 1.114152
[1] -0.1214943
[1] -0.2898258
```

This loop takes the `i` variable and in each iteration of the loop gives it values 1, 2, 3, ..., 10, executes the code within the curly braces, and then the loop exits.

The following three loops all have the same behavior.

```
x <- c("a", "b", "c", "d")

for(i in 1:4) {
        ## Print out each element of 'x'
        print(x[i])
}
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

The `seq_along()` function is commonly used in conjunction with for loops in order to generate an integer sequence based on the length of an object (in this case, the object `x`).

```
## Generate a sequence based on length of 'x'
for(i in seq_along(x)) {
        print(x[i])
}
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

It is not necessary to use an index-type variable.

```
for(letter in x) {
        print(letter)
}
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

For one line loops, the curly braces are not strictly necessary.

```
for(i in 1:4) print(x[i])
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

However, curly braces are sometimes useful even for one-line loops, because that way if you decide to expand the loop to multiple lines, you won't be burned because you forgot to add curly braces (and you *will* be burned by this).

Nested `for` loops

`for` loops can be nested inside of each other.

```
x <- matrix(1:6, 2, 3)

for(i in seq_len(nrow(x))) {
        for(j in seq_len(ncol(x))) {
                print(x[i, j])
        }
}
```

Nested loops are commonly needed for multidimensional or hierarchical data structures (e.g. matrices, lists). Be careful with nesting though. Nesting beyond 2 to 3 levels often makes it difficult to read or understand the code. If you find yourself in need of a large number of nested loops, you may want to break up the loops by using functions (discussed later).

`next`, `break`

`next` is used to skip an iteration of a loop.

```
for(i in 1:100) {
        if(i <= 20) {
                ## Skip the first 20 iterations
                next
        }
        ## Do something here
}
```

`break` is used to exit a loop immediately, regardless of what iteration the loop may be on.

```
for(i in 1:100) {
      print(i)

      if(i > 20) {
              ## Stop loop after 20 iterations
              break
      }
}
```

Summary

- Control structures like `if-else` and `for` allow you to control the flow of an R program.
- Control structures mentioned here are primarily useful for writing programs; for command-line interactive work, the "apply" functions are typically more useful.

2.2 Functions

The learning objectives of the section are:

- Write a function that abstracts a single concept/procedure

The development of a functions in R represents the next level of R programming, beyond executing commands at the command line and writing scripts containing multiple R expressions. When writing R functions, one has to consider the following things:

1. Functions are used to **encapsulate** a sequence of expressions that are executed together to achieve a specific goal. A single function typically does “one thing well”—often taking some input and then generating output that can potentially be handed off to another function for further processing. Drawing the lines where functions begin and end is a key skill for writing functions. When writing a function, it’s important to ask yourself *what do I want to encapsulate*?
2. There is going to be a **user** who will desire the ability to modify certain aspects of your code to match their specific needs or application. Aspects of your code that can be modified often become *function arguments* that can be specified by the user. This user can range from yourself (at a later date) to people you have never met using your code for purposes you never dreamed of. When writing any function it’s important to ask *what will the user want to modify in this function*? Ultimately, the answer to this question will lead to the function’s **interface**.

Code

Often we start out analyzing data by writing straight R code at the console. This code is designed to accomplish a single task—whatever it is that we are trying to do *right now*. For example, consider the following code that operates on download logs published by RStudio from their mirror of the Comprehensive R Archive Network (CRAN). This code counts the number of times the `filehash` package was downloaded on July 20, 2016.

```
library(readr)
library(dplyr)

## Download data from RStudio (if we haven't already)
if(!file.exists("data/2016-07-20.csv.gz")) {
        download.file("http://cran-logs.rstudio.com/2016/2016-07-20.csv.gz",
                      "data/2016-07-20.csv.gz")
}
cran <- read_csv("data/2016-07-20.csv.gz", col_types = "ccicccccci")
cran %>% filter(package == "filehash") %>% nrow
[1] 179
```

This computation is fairly straightforward and if one were only interested in knowing the number of downloads for this package on this day, there would be little more to say about the code. However, there are a few aspects of this code that one might want to modify or expand on:

- the **date**: this code only reads data for July 20, 2016. But what about data from other days? Note that we would first need to obtain that data if we were interested in knowing download statistics from other days.
- the **package**: this code only returns the number of downloads for the `filehash` package. However, there are many other packages on CRAN and we may want to know how many times these other packages were downloaded.

Once we've identified which aspects of a block of code we might want to modify or vary, we can take those things and abstract them to be arguments of a function.

Function interface

The following function has two arguments:

- `pkgname`, the name of the package as a character string
- `date`, a character string indicating the date for which you want download statistics, in year-month-day format

Given the date and package name, the function downloads the appropriate download logs from the RStudio server, reads the CSV file, and then returns the number of downloads for the package.

```r
library(dplyr)
library(readr)

## pkgname: package name (character)
## date: YYYY-MM-DD format (character)
num_download <- function(pkgname, date) {
        ## Construct web URL
        year <- substr(date, 1, 4)
        src <- sprintf("http://cran-logs.rstudio.com/%s/%s.csv.gz",
                       year, date)

        ## Construct path for storing local file
        dest <- file.path("data", basename(src))

        ## Don't download if the file is already there!
        if(!file.exists(dest))
                download.file(src, dest, quiet = TRUE)

        cran <- read_csv(dest, col_types = "ccicccccci", progress = FALSE)
        cran %>% filter(package == pkgname) %>% nrow
}
```

Now we can call our function using whatever date or package name we choose.

```
num_download("filehash", "2016-07-20")
[1] 179
```

We can look up the downloads for a different package on a different day.

```
num_download("Rcpp", "2016-07-19")
[1] 13572
```

Note that for this date, the CRAN log file had to be downloaded separately because it had not yet been downloaded.

Default values

The way that the `num.download()` function is currently specified, the user must enter the date and package name each time the function is called. However, it may be that there is a logical "default date" for which we always want to know the number of downloads, for any package. We can set a **default value** for the date argument, for example, to be July 20, 2016. In that case, if the `date` argument is not explicitly set by the user, the function can use the default value. The revised function might look as follows:

```
num_download <- function(pkgname, date = "2016-07-20") {
        year <- substr(date, 1, 4)
        src <- sprintf("http://cran-logs.rstudio.com/%s/%s.csv.gz",
                       year, date)
        dest <- file.path("data", basename(src))
        if(!file.exists(dest))
                download.file(src, dest, quiet = TRUE)
        cran <- read_csv(dest, col_types = "ccicccccci", progress = FALSE)
        cran %>% filter(package == pkgname) %>% nrow
}
```

Now we can call the function in the following manner. Notice that we do not specify the `date` argument.

```
num_download("Rcpp")
[1] 14761
```

Default values play a critical role in R functions because R functions are often called *interactively*. When using R in interactive mode, it can be a pain to have to specify the value of every argument in every instance of calling the function. Sometimes we want to call a function multiple times while varying a single argument (keeping the other arguments at a sensible default).

Also, function arguments have a tendency to proliferate. As functions mature and are continuously developed, one way to add more functionality is to increase the number of arguments. But if these new arguments do not have sensible default values, then users will generally have a harder time using the function.

As a function author, you have tremendous influence over the user's behavior by specifying defaults, so take care in choosing them. However, just note that a judicious use of default values can greatly improve the user experience with respect to your function.

Re-factoring code

Now that we have a function written that handles the task at hand in a more general manner (i.e. it can handle any package and any date), it is worth taking a closer look at the function and asking whether it is written in the most useful possible manner. In particular, it could be argued that this function does too many things:

1. Construct the path to the remote and local log file
2. Download the log file (if it doesn't already exist locally)
3. Read the log file into R
4. Find the package and return the number of downloads

It might make sense to abstract the first two things on this list into a separate function. For example, we could create a function called `check_for_logfile()` to see if we need to download the log file and then `num_download()` could call this function.

```
check_for_logfile <- function(date) {
        year <- substr(date, 1, 4)
        src <- sprintf("http://cran-logs.rstudio.com/%s/%s.csv.gz",
                       year, date)
        dest <- file.path("data", basename(src))
        if(!file.exists(dest)) {
                val <- download.file(src, dest, quiet = TRUE)
                if(!val)
                        stop("unable to download file ", src)
        }
        dest
}
```

This file takes the original download code from `num_download()` and adds a bit of error checking to see if `download.file()` was successful (if not, an error is thrown with `stop()`).

Now the `num_download()` function is somewhat simpler.

```
num_download <- function(pkgname, date = "2016-07-20") {
        dest <- check_for_logfile(date)
        cran <- read_csv(dest, col_types = "ccicccccci", progress = FALSE)
        cran %>% filter(package == pkgname) %>% nrow
}
```

In addition to being simpler to read, another key difference is that the `num_download()` function does not need to know anything about downloading or URLs or files. All it knows is that there is a function `check_for_logfile()` that just deals with getting the data to your computer. From there, we can just read the data with `read_csv()` and get the information we need. This is the value of abstraction and writing functions.

Dependency Checking

The `num_downloads()` function depends on the `readr` and `dplyr` packages. Without them installed, the function won't run. Sometimes it is useful to check to see that the needed packages are installed so that a useful error message (or other behavior) can be provided for the user.

We can write a separate function to check that the packages are installed.

```
check_pkg_deps <- function() {
        if(!require(readr)) {
                message("installing the 'readr' package")
                install.packages("readr")
        }
        if(!require(dplyr))
                stop("the 'dplyr' package needs to be installed first")
}
```

There are a few things to note about this function. First, it uses the `require()` function to attempt to load the `readr` and `dplyr` packages. The `require()` function is similar to `library()`, however `library()` stops with an error if the package cannot be loaded whereas `require()` returns `TRUE` or `FALSE` depending on whether the package can be loaded or not. For both functions, if the package is available, it is loaded and attached to the `search()` path.

Typically, `library()` is good for interactive work because you usually can't go on without a specific package (that's why you're loading it in the first place!). On the other hand, `require()` is good for programming because you may want to engage in different behaviors depending on which packages are not available.

For example, in the above function, if the `readr` package is not available, we go ahead and install the package for the user (along with providing a message). However, if we cannot load the `dplyr` package we throw an error. This distinction in behaviors for `readr` and `dplyr` is a bit arbitrary in this case, but it illustrates the flexibility that is afforded by using `require()` versus `library()`.

Now, our updated function can check for package dependencies.

```
num_download <- function(pkgname, date = "2016-07-20") {
        check_pkg_deps()
        dest <- check_for_logfile(date)
        cran <- read_csv(dest, col_types = "ccicccccci", progress = FALSE)
        cran %>% filter(package == pkgname) %>% nrow
}
```

Vectorization

One final aspect of this function that is worth noting is that as currently written it is not *vectorized*. This means that each argument must be a single value—a single package name and a single date. However, in R, it is a common paradigm for functions to take vector arguments and for those functions to return vector or list results. Often, users are bitten by unexpected behavior because a function is assumed to be vectorized when it is not.

One way to vectorize this function is to allow the `pkgname` argument to be a character vector of package names. This way we can get download statistics for multiple packages with a single function call. Luckily, this is fairly straightforward to do. The two things we need to do are

1. Adjust our call to `filter()` to grab rows of the data frame that fall within a vector of package names
2. Use a `group_by() %>% summarize()` combination to count the downloads *for each* package.

```
## 'pkgname' can now be a character vector of names
num_download <- function(pkgname, date = "2016-07-20") {
        check_pkg_deps()
        dest <- check_for_logfile(date)
        cran <- read_csv(dest, col_types = "ccicccccci", progress = FALSE)
        cran %>% filter(package %in% pkgname) %>%
                group_by(package) %>%
                summarize(n = n())
}
```

Now we can call the following

```
num_download(c("filehash", "weathermetrics"))
# A tibble: 2 × 2
         package     n
           <chr> <int>
1       filehash   179
2 weathermetrics     7
```

Note that the output of `num_download()` has changed. While it previously returned an integer vector, the vectorized function returns a data frame. If you are authoring a function that is used by many people, it is usually wise to give them some warning before changing the nature of the output.

Vectorizing the `date` argument is similarly possible, but it has the added complication that for each date you need to download another log file. We leave this as an exercise for the reader.

Argument Checking

Checking that the arguments supplied by the reader are proper is a good way to prevent confusing results or error messages from occurring later on in the function. It is also a useful way to enforce documented requirements for a function.

In this case, the `num_download()` function is expecting both the `pkgname` and `date` arguments to be character vectors. In particular, the `date` argument should be a character vector of length 1. We can check the class of an argument using `is.character()` and the length using the `length()` function.

The revised function with argument checking is as follows.

```
num_download <- function(pkgname, date = "2016-07-20") {
        check_pkg_deps()

        ## Check arguments
        if(!is.character(pkgname))
                stop("'pkgname' should be character")
        if(!is.character(date))
                stop("'date' should be character")
        if(length(date) != 1)
                stop("'date' should be length 1")

        dest <- check_for_logfile(date)
        cran <- read_csv(dest, col_types = "ccicccccci",
                         progress = FALSE)
        cran %>% filter(package %in% pkgname) %>%
                group_by(package) %>%
                summarize(n = n())
}
```

Note that here, we chose to `stop()` and throw an error if the argument was not of the appropriate type. However, an alternative would have been to simply coerce the argument to be of character type using the `as.character()` function.

```
num_download("filehash", c("2016-07-20", "2016-0-21"))
Error in num_download("filehash", c("2016-07-20", "2016-0-21")): 'date' should be length 1
```

R package

R packages are collections of functions that together allow one to conduct a series of related operations. We will not go into detail about R packages here, but we bring them up only to indicate that they are the natural evolution of writing many functions. R packages similarly have an interface or API which specifies to the user what functions he/she can call in their own code. The development and maintenance of R packages is the major focus of the next chapter.

When Should I Write a Function?

Deciding when to write a function depends on the context in which you are programming in R. For a one-off type of activity, it's probably not worth considering the design of a function or set of functions. However, in our experience, there are relatively few one-off scenarios. In particular, such a scenario implies that whatever you did worked on the very first try.

In reality, we often have to repeat certain tasks or we have to share code with others. Sometimes those "other people" are simply ourselves 3 months later. As the great Karl Broman once famously said

> Your closest collaborator is you six months ago, but you don't reply to emails.

This comment relates to the general question of whether some code will ever have any **users**, including yourself later on. If the code will likely have more than one user, they will benefit from the abstraction and simplification afforded by encapsulating the code in functions and providing a clean interface.

In Roger's book, *Executive Data Science*, he writes about when to write a function:

- If you're going to do something **once** (that does happen on occasion), just write some code and *document it very well*. The important thing is that you want to make sure that you understand what the code does, and so that requires both writing the code well and writing documentation. You want to be able to reproduce it later on if you ever come back to it, or if someone else comes back to it.
- If you're going to do something **twice**, write a function. This allows you to abstract a small piece of code, and it forces you to define an interface, so you have well defined inputs and outputs.
- If you're going to do something **three** times or more, you should think about writing a small package. It doesn't have to be commercial level software, but a small package which encapsulates the set of operations that you're going to be doing in a given analysis. It's also important to write some real documentation so that people can understand what's supposed to be going on, and can apply the software to a different situation if they have to.

Summary

Developing functions is a key aspect of programming in R and typically involves a bottom-up process.

- Code is written to accomplish a specific task or a specific instance of a task.

- The code is examined to identify key aspects that may be modified by other users; these aspects are abstracted out of the code and made into arguments of a function.
- Functions are written to accomplish more general versions of a task; specific instances of the task are indicated by setting values of function arguments.
- Function code can be re-factored to provide better modularity and to divide functions into specific sub-tasks.
- Functions can be assembled and organized into R packages.

2.3 Functional Programming

The learning objectives of the section are:

- Describe functional programming concepts
- Write functional programming code using the `purrr` package

What is Functional Programming?

Functional programming is a programming philosophy based on lambda calculus. Lambda calculus was created by Alonzo Church, the PhD adviser to Alan Turing who is known for his role in cracking the encryption of the Nazi's Enigma machine during World War Two. Functional programming has been a popular approach ever since it helped bring down the Third Reich.

Functional programming concentrates on four constructs:

1. Data (numbers, strings, etc)
2. Variables (function arguments)
3. Functions
4. Function Applications (evaluating functions given arguments and/or data)

By now you're used to treating variables inside of functions as data, whether they're values like numbers and strings, or they're data structures like lists and vectors. With functional programming you can also consider the possibility that you can provide a function as an argument to another function, and a function can return another function as its result.

If you've used functions like `sapply()` or `args()` then it's easy to imagine how functions as arguments to other functions can be used. In the case of `sapply()` the provided function is applied to data, and in the case of `args()` information about the function is returned. What's rarer to see is a function that returns a function when it's evaluated. Let's look at a small example of how this can work:

```
adder_maker <- function(n){
  function(x){
    n + x
  }
}

add2 <- adder_maker(2)
add3 <- adder_maker(3)

add2(5)
[1] 7
add3(5)
[1] 8
```

In the example above the function `adder_maker()` returns a function with no name. The function returned adds `n` to its only argument `x`.

Core Functional Programming Functions

There are groups of functions that are essential for functional programming. In most cases they take a function and a data structure as arguments, and that function is applied to that data structure in some way. The `purrr` library contains many of these functions and we'll be using it throughout this section. Function programming is concerned mostly with lists and vectors. I may refer to just lists or vectors, but you should know that what applies for lists generally applies for vectors and vice-versa.

Map

The map family of functions applies a function to the elements of a data structure, usually a list or a vector. The function is evaluated once for each element of the vector with the vector element as the first argument to the function. The return value is the same kind if data structure (a list or vector) but with every element replaced by the result of the function being evaluated with the corresponding element as the argument to the function. In the `purrr` package the `map()` function returns a list, while the `map_lgl()`, `map_chr()`, and `map_dbl()` functions return vectors of logical values, strings, or numbers respectively. Let's take a look at a few examples:

```
library(purrr)

map_chr(c(5, 4, 3, 2, 1), function(x){
  c("one", "two", "three", "four", "five")[x]
})
[1] "five"  "four"  "three" "two"   "one"

map_lgl(c(1, 2, 3, 4, 5), function(x){
  x > 3
})
[1] FALSE FALSE FALSE  TRUE  TRUE
```

Think about evaluating each function above with just one of the arguments in the specified numeric vector, and then combining all of those function results into one vector.

The `map_if()` function takes as its arguments a list or vector containing data, a predicate function, and then a function to be applied. A predicate function is a function that returns `TRUE` or `FALSE` for each element in the provided list or vector. In the case of `map_if()`: if the predicate functions evaluates to `TRUE`, then the function is applied to the corresponding vector element, however if the predicate function evaluates to `FALSE` then the function is not applied. The `map_if()` function always returns a list, so I'm piping the result of `map_if()` to `unlist()` so it look prettier:

```
map_if(1:5, function(x){
              x %% 2 == 0
            },
            function(y){
              y^2
            }) %>% unlist()
[1]  1  4  3 16  5
```

Notice how only the even numbers are squared, while the odd numbers are left alone.

The `map_at()` function only applies the provided function to elements of a vector specified by their indexes. `map_at()` always returns a list so like before I'm piping the result to `unlist()`:

```
map_at(seq(100, 500, 100), c(1, 3, 5), function(x){
  x - 10
}) %>% unlist()
[1]  90 200 290 400 490
```

Like we expected to happen the providied function is only applied to the first, third, and fifth element of the vector provided.

In each of the examples above we have only been mapping a function over one data structure, however you can map a function over two data structures with the `map2()` family of functions. The first two arguments should be two vectors of the same length, followed by a function which will be evaluated with an element of the first vector as the first argument and an element of the second vector as the second argument. For example:

```
map2_chr(letters, 1:26, paste)
 [1] "a 1"  "b 2"  "c 3"  "d 4"  "e 5"  "f 6"  "g 7"  "h 8"  "i 9"  "j 10"
[11] "k 11" "l 12" "m 13" "n 14" "o 15" "p 16" "q 17" "r 18" "s 19" "t 20"
[21] "u 21" "v 22" "w 23" "x 24" "y 25" "z 26"
```

The `pmap()` family of functions is similar to `map2()`, however instead of mapping across two vectors or lists, you can map across any number of lists. The list argument is a list of lists that the function will map over, followed by the function that will applied:

```
pmap_chr(list(
  list(1, 2, 3),
  list("one", "two", "three"),
  list("uno", "dos", "tres")
), paste)
[1] "1 one uno"    "2 two dos"    "3 three tres"
```

Mapping is a powerful technique for thinking about how to apply computational operations to your data.

Reduce

List or vector reduction iteratively combines the first element of a vector with the second element of a vector, then that combined result is combined with the third element of the vector, and so on until the end of the vector is reached. The function to be applied should take at least two arguments. Where mapping returns a vector or a list, reducing should return a single value. Some examples using `reduce()` are illustrated below:

```
reduce(c(1, 3, 5, 7), function(x, y){
  message("x is ", x)
  message("y is ", y)
  message("")
  x + y
})
x is 1
y is 3

x is 4
y is 5

x is 9
y is 7

[1] 16
```

On the first iteration `x` has the value 1 and `y` has the value `3`, then the two values are combined (they're added together). On the second iteration `x` has the value of the result from the first iteration (4) and y has the value of the third element in the provided numeric vector (5). This process is repeated for each iteration. Here's a similar example using string data:

```
reduce(letters[1:4], function(x, y){
  message("x is ", x)
  message("y is ", y)
  message("")
  paste0(x, y)
})
x is a
y is b

x is ab
y is c

x is abc
y is d

[1] "abcd"
```

By default `reduce()` starts with the first element of a vector and then the second element and so on. In contrast the `reduce_right()` function starts with the last element of a vector and then proceeds to the second to last element of a vector and so on:

```
reduce_right(letters[1:4], function(x, y){
  message("x is ", x)
  message("y is ", y)
  message("")
  paste0(x, y)
})
x is d
y is c

x is dc
y is b

x is dcb
y is a

[1] "dcba"
```

Search

You can search for specific elements of a vector using the `contains()` and `detect()` functions. `contains()` will return `TRUE` if a specified element is present in a vector, otherwise it returns `FALSE`:

```
contains(letters, "a")
[1] TRUE
contains(letters, "A")
[1] FALSE
```

The `detect()` function takes a vector and a predicate function as arguments and it returns the first element of the vector for which the predicate function returns `TRUE`:

```
detect(20:40, function(x){
  x > 22 && x %% 2 == 0
})
[1] 24
```

The `detect_index()` function takes the same arguments, however it returns the index of the provided vector which contains the first element that satisfies the predicate function:

```
detect_index(20:40, function(x){
  x > 22 && x %% 2 == 0
})
[1] 5
```

Filter

The group of functions that includes `keep()`, `discard()`, `every()`, and `some()` are known as filter functions. Each of these functions takes a vector and a predicate function. For `keep()` only the elements of the vector that satisfy the predicate function are returned while all other elements are removed:

```
keep(1:20, function(x){
  x %% 2 == 0
})
 [1]  2  4  6  8 10 12 14 16 18 20
```

The `discard()` function works similarly, it only returns elements that don't satisfy the predicate function:

```
discard(1:20, function(x){
  x %% 2 == 0
})
 [1]  1  3  5  7  9 11 13 15 17 19
```

The `every()` function returns `TRUE` only if every element in the vector satisfies the predicate function, while the `some()` function returns `TRUE` if at least one element in the vector satisfies the predicate function:

```
every(1:20, function(x){
  x %% 2 == 0
})

some(1:20, function(x){
  x %% 2 == 0
})
```

Compose

Finally, the `compose()` function combines any number of functions into one function:

```
n_unique <- compose(length, unique)
# The composition above is the same as:
# n_unique <- function(x){
#   length(unique(x))
# }

rep(1:5, 1:5)
 [1] 1 2 2 3 3 3 4 4 4 4 5 5 5 5 5

n_unique(rep(1:5, 1:5))
[1] 5
```

Functional Programming Concepts

Partial Application

Partial application of functions can allow functions to behave a little like data structures. Using the `partial()` function from the `purrr` package you can specify some of the arguments of a function, and then `partial()` will return a function that only takes the unspecified arguments. Let's take a look at a simple example:

```
library(purrr)

mult_three_n <- function(x, y, z){
  x * y * z
}

mult_by_15 <- partial(mult_three_n, x = 3, y = 5)

mult_by_15(z = 4)
[1] 60
```

By using partial application you can bind some data to the arguments of a function before using that function elsewhere.

Side Effects

Side effects of functions occur whenever a function interacts with the "outside world" – reading or writing data, printing to the console, and displaying a graph are all side effects. The results of side effects are one of the main motivations for writing code in the first place! Side effects can be tricky to handle though, since the order in which functions with side effects are executed often matters and there are variables that are external to the program (the relative location of some data). If you want to evaluate a function across a data structure you should use the `walk()` function from `purrr`. Here's a simple example:

```
library(purrr)

walk(c("Friends, Romans, countrymen,",
       "lend me your ears;",
       "I come to bury Caesar,",
       "not to praise him."), message)
Friends, Romans, countrymen,
lend me your ears;
I come to bury Caesar,
not to praise him.
```

Recursion

Recursion is very powerful tool, both mentally and in software development, for solving problems. Recursive functions have two main parts: a few easy to solve problems called "base cases," and then a case for more complicated problems where **the function is called inside of itself**. The central philosophy of recursive programming is that problems can be broken down into simpler parts, and then combining those simple answers results in the answer to a complex problem.

Imagine you wanted to write a function that adds together all of the numbers in a vector. You could of course accomplish this with a loop:

```
vector_sum_loop <- function(v){
  result <- 0
  for(i in v){
    result <- result + i
  }
  result
}

vector_sum_loop(c(5, 40, 91))
[1] 136
```

You could also think about how to solve this problem recursively. First ask yourself: what's the base case of finding the sum of a vector? If the vector only contains one element, then the sum is just the value of that element. In the more complex case the vector has more than one element. We can remove the first element of the vector, but then what should we do with the rest of the vector? Thankfully we have a function for computing the sum of all of the elements of a vector because we're writing that function right now! So we'll add the value of the first element of the vector to whatever the cumulative sum is of the rest of the vector. The resulting function is illustrated below:

```
vector_sum_rec <- function(v){
  if(length(v) == 1){
    v
  } else {
    v[1] + vector_sum_rec(v[-1])
  }
}

vector_sum_rec(c(5, 40, 91))
[1] 136
```

Another useful exercise for thinking about applications for recursion is computing the Fibonacci sequence. The Fibonacci sequence is a sequence of integers that starts: 0, 1, 1, 2, 3, 5, 8 where each proceeding integer is the sum of the previous two integers. This fits into a recursive mental framework very nicely since each subsequent number depends on the previous two numbers.

Let's write a function to computues the nth digit of the Fibonacci sequence such that the first number in the sequence is 0, the second number is 1, and then all proceeding numbers are the sum of the n - 1 and the n - 2 Fibonacci number. It is immediately evident that there are three base cases:

1. n must be greater than 0.
2. When n is equal to 1, return 0.
3. When n is equal to 2, return 1.

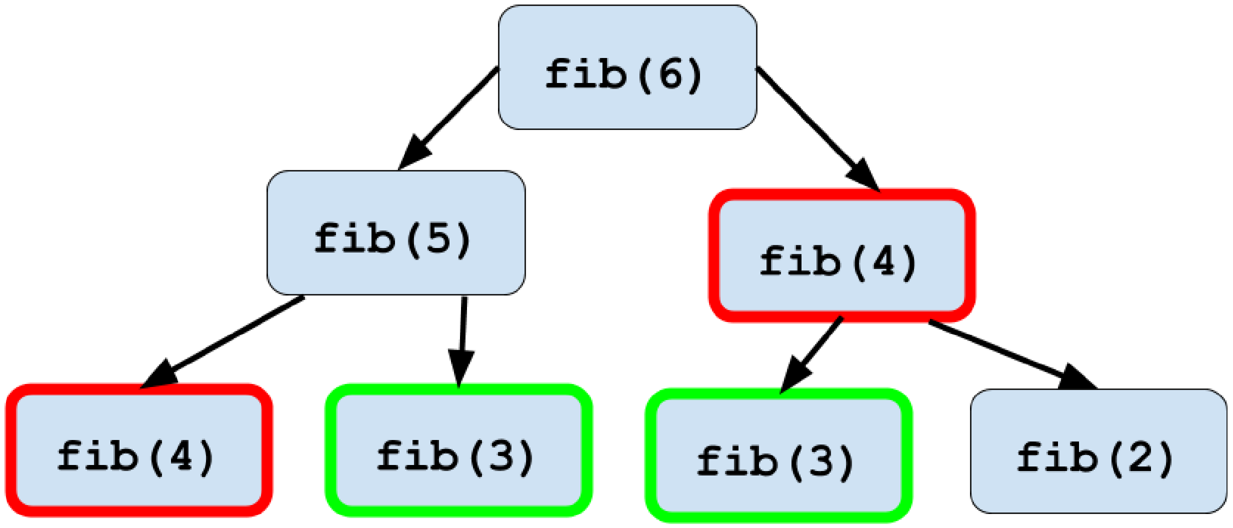
And then the recursive case:

- Otherwise return the sum of the n - 1 Fibonacci number and the n - 2 Fibonacci number.

Let's turn those words into code:

```
fib <- function(n){
  stopifnot(n > 0)
  if(n == 1){
    0
  } else if(n == 2){
    1
  } else {
    fib(n - 1) + fib(n - 2)
  }
}

fib(1)
[1] 0
fib(2)
[1] 1
fib(3)
[1] 1
fib(4)
[1] 2
fib(5)
[1] 3
fib(6)
[1] 5
fib(7)
[1] 8

map_dbl(1:12, fib)
 [1]  0  1  1  2  3  5  8 13 21 34 55 89
```

Looks like it's working well! There is one optimization that we could apply here which comes up in recursive programming often. When you execute the function `fib(6)`, within that function you'll execute `fib(5)` and `fib(4)`. Then within the execution of `fib(5)`, `fib(4)` will be executed again. An illustration of this phenomenon is below:



Memoization of fib() function

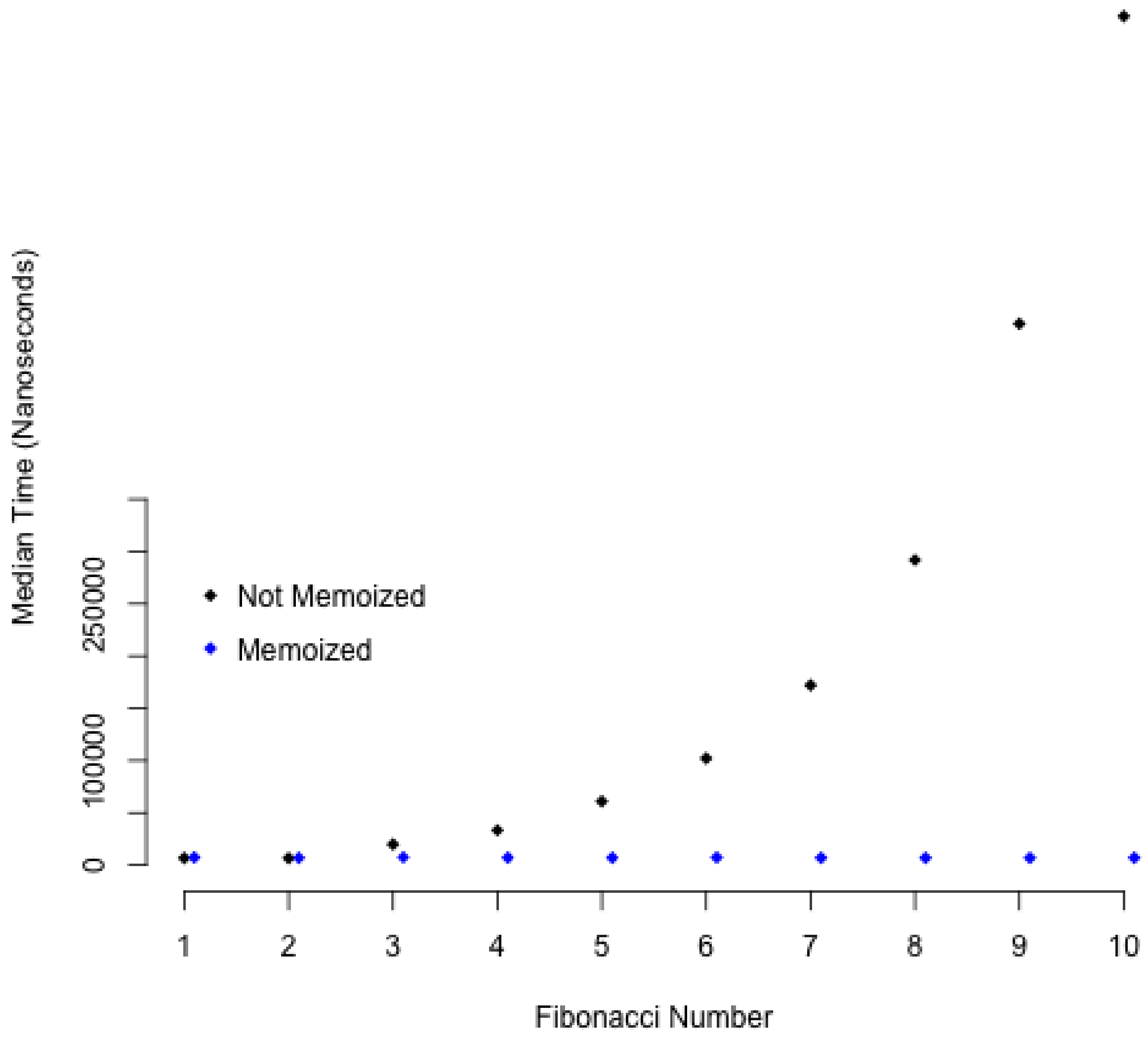
This duplication of computation slows down your program significantly as you calculate larger numbers in the Fibonacci sequence. Thankfully you can use a technique called memoization in order to speed this computation up. Memoization stores the value of each calculated Fibonacci number in table so that once a number is calculated you can look it up instead of needing to recalculate it!

Below is an example of a function that can calculate the first 25 Fibonacci numbers. First we'll create a very simple table which is just a vector containing 0, 1, and then 23 `NA`s. First the `fib_mem()` function will check if the number is in the table, and if it is then it is returned. Otherwise the Fibonacci number is recursively calculated and stored in the table. Notice that we're using the complex assignment operator `<<-` in order to modify the table outside the scope of the function. You'll learn more about the complex operator in the section titled *Expressions & Environments*.

```
fib_tbl <- c(0, 1, rep(NA, 23))

fib_mem <- function(n){
  stopifnot(n > 0)

  if(!is.na(fib_tbl[n])){
    fib_tbl[n]
  } else {
    fib_tbl[n - 1] <<- fib_mem(n - 1)
    fib_tbl[n - 2] <<- fib_mem(n - 2)
    fib_tbl[n - 1] + fib_tbl[n - 2]
  }
}

map_dbl(1:12, fib_mem)
 [1]  0  1  1  2  3  5  8 13 21 34 55 89
```

It works! But is it any faster than the original `fib()`? Below I'm going to use the `microbenchmark` package in order assess whether `fib()` or `fib_mem()` is faster:

```
library(purrr)
library(microbenchmark)
library(tidyr)
library(magrittr)
library(dplyr)

fib_data <- map(1:10, function(x){microbenchmark(fib(x), times = 100)$time})
names(fib_data) <- paste0(letters[1:10], 1:10)
fib_data <- as.data.frame(fib_data)

fib_data %<>%
  gather(num, time) %>%
  group_by(num) %>%
  summarise(med_time = median(time))

memo_data <- map(1:10, function(x){microbenchmark(fib_mem(x))$time})
names(memo_data) <- paste0(letters[1:10], 1:10)
memo_data <- as.data.frame(memo_data)

memo_data %<>%
  gather(num, time) %>%
  group_by(num) %>%
  summarise(med_time = median(time))

plot(1:10, fib_data$med_time, xlab = "Fibonacci Number", ylab = "Median Time (Nanoseconds)",
     pch = 18, bty = "n", xaxt = "n", yaxt = "n")
axis(1, at = 1:10)
axis(2, at = seq(0, 350000, by = 50000))
points(1:10 + .1, memo_data$med_time, col = "blue", pch = 18)
legend(1, 300000, c("Not Memoized", "Memoized"), pch = 18,
       col = c("black", "blue"), bty = "n", cex = 1, y.intersp = 1.5)
```



Speed comparison of memoization

As you can see as higher Fibonacci numbers are calculated the time it takes to calculate a number with `fib()` grows exponentially, while the time it takes to do the same task with `fib_mem()` stays constant.

Summary

- Functional programming is based on lambda calculus.
- This approach concentrates on data, variables, functions, and function applications.
- It's possible for functions to be able to return other functions.

- The core functional programming concepts can be summarized in the following categories: map, reduce, search, filter, and compose.
- Partial application of functions allows functions to be used like data sctructures.
- Side effects are difficult to debug although they motivate a huge fraction of computer programming.
- The most important part of understanding recursion is understanding recursion.

2.4 Expressions & Environments

The learning objectives of this section are:

- Manipulate R expressions to “compute on the language”
- Describe the semantics of R environments

Expressions

Expressions are encapsulated operations that can be executed by R. This may sound complicated, but using expressions allows you manipulate code with code! You can create an expression using the `quote()` function. For that function’s argument, just type whatever you would normally type into the R console. For example:

```
two_plus_two <- quote(2 + 2)
two_plus_two
2 + 2
```

You can execute this expressions using the `eval()` function:

```
eval(two_plus_two)
[1] 4
```

You might encounter R code that is stored as a string that you want to evaluate with `eval()`. You can use `parse()` to transform a string into an expression:

```
tpt_string <- "2 + 2"

tpt_expression <- parse(text = tpt_string)

eval(tpt_expression)
[1] 4
```

You can reverse this process and transform an expression into a string using `deparse()`:

```
deparse(two_plus_two)
[1] "2 + 2"
```

One interesting feature about expressions is that you can access and modify their contents like you a `list()`. This means that you can change the values in an expression, or even the function being executed in the expression before it is evaluated:

```
sum_expr <- quote(sum(1, 5))
eval(sum_expr)
[1] 6
sum_expr[[1]]
sum
sum_expr[[2]]
[1] 1
sum_expr[[3]]
[1] 5
sum_expr[[1]] <- quote(paste0)
sum_expr[[2]] <- quote(4)
sum_expr[[3]] <- quote(6)
eval(sum_expr)
[1] "46"
```

You can compose expressions using the `call()` function. The first argument is a string containing the name of a function, followed by the arguments that will be provided to that function.

```
sum_40_50_expr <- call("sum", 40, 50)
sum_40_50_expr
sum(40, 50)
eval(sum_40_50_expr)
[1] 90
```

You can capture the the expression an R user typed into the R console when they executed a function by including `match.call()` in the function the user executed:

```
return_expression <- function(...){
  match.call()
}

return_expression(2, col = "blue", FALSE)
return_expression(2, col = "blue", FALSE)
```

You could of course then manipulate this expression inside of the function you're writing. The exmaple below first uses `match.call()` to capture the expression that the user entered. The first argument of the function is then extracted an evaluated. If the first expressions is a number, then a string is returned describing the first argument, otherwise the string `"The first argument is not numeric."` is returned.

```
first_arg <- function(...){
  expr <- match.call()
  first_arg_expr <- expr[[2]]
  first_arg <- eval(first_arg_expr)
  if(is.numeric(first_arg)){
    paste("The first argument is", first_arg)
  } else {
    "The first argument is not numeric."
  }
}

first_arg(2, 4, "seven", FALSE)
[1] "The first argument is 2"

first_arg("two", 4, "seven", FALSE)
[1] "The first argument is not numeric."
```

Expressions are a powerful tool for writing R programs that can manipulate other R programs.

Environments

Environments are data structures in R that have special properties with regard to their role in how R code is executed and how memory in R is organized. You may not realize it but you're probably already familiar with one environment called the global environment. Environments formalize relationships between variable names and values. When you enter `x <- 55` into the R console what you're saying is: assign the value of 55 to a variable called `x`, and store this assignment in the global environment. The global environment is therefore where most R users do most of their programming and analysis.

You can create a new environment using `new.env()`. You can assign variables in that environment in a similar way to assigning a named element of a list, or you can use `assign()`. You can retrieve the value of a variable just like you would retrieve the named element of a list, or you can use `get()`. Notice that `assign()` and `get()` are opposites:

```
my_new_env <- new.env()
my_new_env$x <- 4
my_new_env$x
[1] 4

assign("y", 9, envir = my_new_env)
get("y", envir = my_new_env)
[1] 9
my_new_env$y
[1] 9
```

You can get all of the variable names that have been assigned in an environment using `ls()`, you can remove an association between a variable name and a value using `rm()`, and you can check if a variable name has been assigned in an environment using `exists()`:

```
ls(my_new_env)
[1] "x" "y"
rm(y, envir = my_new_env)
exists("y", envir = my_new_env)
[1] TRUE
exists("x", envir = my_new_env)
[1] TRUE
my_new_env$x
[1] 4
my_new_env$y
NULL
```

Environments are organized in parent/child relationships such that every environment keeps track of its parent, but parents are unaware of which environments are their children. Usually the relationships between environments is not something you should try to directly control. You can see the parents of the global environment using the `search()` function:

```
search()
 [1] ".GlobalEnv"               "package:magrittr"
 [3] "package:tidyr"            "package:microbenchmark"
 [5] "package:purrr"            "package:dplyr"
 [7] "package:readr"            "package:parallel"
 [9] "package:knitr"            "package:stats"
[11] "package:graphics"         "package:grDevices"
[13] "package:utils"            "package:datasets"
[15] "Autoloads"                "package:base"
```

As you can see package:magrittr is the parent of .GlobalEnv, and package:tidyr is parent of package:magrittr, and so on. In general the parent of .GlobalEnv is always the last package that was loaded using `library()`. Notice that after I load the `ggplot2` package, that package becomes the parent of .GlobalEnv:

```
library(ggplot2)
search()
 [1] ".GlobalEnv"             "package:ggplot2"
 [3] "package:magrittr"       "package:tidyr"
 [5] "package:microbenchmark" "package:purrr"
 [7] "package:dplyr"          "package:readr"
 [9] "package:parallel"       "package:knitr"
[11] "package:stats"          "package:graphics"
[13] "package:grDevices"      "package:utils"
[15] "package:datasets"       "Autoloads"
[17] "package:base"
```

Execution Environments

Although there may be several cases where you need to create a new environment using `new.env()`, you will more often create new environments whenever you execute functions.

An execution environment is an environment that exists temporarily within the scope of a function that is being executed. For example if we have the following code:

```
x <- 10

my_func <- function(){
  x <- 5
  return(x)
}

my_func()
```

What do you think will be the result of `my_func()`? Make your guess and then take a look at the executed code below:

```
x <- 10

my_func <- function(){
  x <- 5
  return(x)
}

my_func()
[1] 5
```

So what exactly is happening above? First the name `x` is bring assigned the value 10 in the global environment. Then the name `my_func` is being assigned the value of the function `function(){x <- 5};return(x)}` in the global environment. When `my_func()` is executed, a new environment is created called the execution environment which only exists while `my_func()` is running. Inside of the execution environment the name `x` is assigned the value 5. When `return()` is executed it looks first in the execution environment for a value that is assigned to `x`. Then the value 5 is returned. In contrast to the situation above, take a look at this variation:

```
x <- 10

another_func <- function(){
  return(x)
}

another_func()
[1] 10
```

In this situation the execution environment inside of `another_func()` does not contain an assignment for the name `x`, so R looks for an assignment in the parent environment of the execution environment which is the global environment. Since `x` is assigned the value 10 in the global environment 10 is returned.

After seeing the cases above you may be curious if it's possible for an execution environment to manipulate the global environment. You're already familiar with the assignment operator `<-`, however you should also be aware that there's another assignment operator called the *complex assignment operator* which looks like `<<-`. You can use the complex assignment operator to re-assign or even create name-value bindings in the global environment from within an execution environment. In this first example, the function `assign1()` will change the value associated with the name `x`:

```
x <- 10
x
[1] 10

assign1 <- function(){
  x <<- "Wow!"
}

assign1()
x
[1] "Wow!"
```

You can see that the value associated with `x` has been changed from 10 to `"Wow!"` in the global environment. You can also use `<<-` to assign names to values that have not been yet been defined in the global environment *from inside a function*:

```
a_variable_name
Error in eval(expr, envir, enclos): object 'a_variable_name' not found
exists("a_variable_name")
[1] FALSE

assign2 <- function(){
  a_variable_name <<- "Magic!"
}

assign2()
exists("a_variable_name")
[1] TRUE
a_variable_name
[1] "Magic!"
```

If you want to see a case for using `<<-` in action, see the section of this book about functional programming and the discussion there about memoization.

Summary

- Expressions are a powerful tool for manipulating and executing R code.
- Environments record associations between names and values.
- Execution environments create a scope for variable names inside of functions.

2.5 Error Handling and Generation

The learning objectives of this section are:

- Implement exception handling routines in R functions

What is an error?

Errors most often occur when code is used in a way that it is not intended to be used. For example adding two strings together produces the following error:

```
"hello" + "world"
Error in "hello" + "world": non-numeric argument to binary operator
```

The `+` operator is essentially a function that takes two numbers as arguments and finds their sum. Since neither `"hello"` nor `"world"` are numbers, the R interpreter produces an error. Errors will stop the execution of your program, and they will (hopefully) print an error message to the R console.

In R there are two other constructs which are related to errors: warnings and messages. Warnings are meant to indicate that something seems to have gone wrong in your program that should be inspected. Here's a simple example of a warning being generated:

```
as.numeric(c("5", "6", "seven"))
Warning: NAs introduced by coercion
[1]  5  6 NA
```

The `as.numeric()` function attempts to convert each string in `c("5", "6", "seven")` into a number, however it is impossible to convert `"seven"`, so a warning is generated. Execution of the code is not halted, and an `NA` is produced for `"seven"` instead of a number.

Messages simply print to the R console, though they are generated by an underlying mechanism that is similar to how errors and warning are generated. Here's a small function that will generate a message:

```
f <- function(){
  message("This is a message.")
}

f()
This is a message.
```

Generating Errors

There are a few essential functions for generating errors, warnings, and messages in R. The `stop()` function will generate an error. Let's generate an error:

```
stop("Something erroneous has occurred!")
```

```
Error: Something erroneous has occurred!
```

If an error occurs inside of a function then the name of that function will appear in the error message:

```
name_of_function <- function(){
  stop("Something bad happened.")
}

name_of_function()
Error in name_of_function(): Something bad happened.
```

The `stopifnot()` function takes a series of logical expressions as arguments and if any of them are false an error is generated specifying which expression is false. Let's take a look at an example:

```
error_if_n_is_greater_than_zero <- function(n){
  stopifnot(n <= 0)
  n
}

error_if_n_is_greater_than_zero(5)
Error: n <= 0 is not TRUE
```

The `warning()` function creates a warning, and the function itself is very similar to the `stop()` function. Remember that a warning does not stop the execution of a program (unlike an error.)

```
warning("Consider yourself warned!")
Warning: Consider yourself warned!
```

Just like errors, a warning generated inside of a function will include the name of the function in which it was generated:

```
make_NA <- function(x){
  warning("Generating an NA.")
  NA
}

make_NA("Sodium")
Warning in make_NA("Sodium"): Generating an NA.
[1] NA
```

Messages are simpler than errors or warnings; they just print strings to the R console. You can issue a message with the `message()` function:

```
message("In a bottle.")
In a bottle.
```

When to generate errors or warnings

Stopping the execution of your program with `stop()` should only happen in the event of a catastrophe - meaning only if it is impossible for your program to continue. If there are conditions that you can anticipate that would cause your program to create an error then you should document those conditions so whoever uses your software is aware. Common failure conditions like providing invalid arguments to a function should be checked at the beginning of your program so that the user can quickly realize something has gone wrong. Checking function inputs is a typical use of the `stopifnot()` function.

You can think of a function as kind of contract between you and the user: if the user provides specified arguments, your program will provide predictable results. Of course it's impossible for you to anticipate all of the potential uses of your program, so the results of executing a function can only be predictable with regard to the type of the result. It's appropriate to create a warning when this contract between you and the user is violated. A perfect example of this situation is the result of `as.numeric(c("5", "6", "seven"))`, which we saw before. The user expects a vector of numbers to be returned as the result of `as.numeric()` but `"seven"` is coerced into being NA, which is not completely intuitive.

R has largely been developed according to the Unix Philosophy (which is further discussed in Chapter 3), which generally discourages printing text to the console unless something unexpected has occurred. Languages that commonly run on Unix systems like C, C++, and Go are rarely used interactively, meaning that they usually underpin computer infrastructure (computers "talking" to other computers). Messages printed to the console are therefore not very useful since nobody will ever read them and it's not straightforward for other programs to capture and interpret them. In contrast R code is frequently executed by human beings in the R console, which serves as an interactive environment between the computer and person at the keyboard. If you think your program should produce a message, make sure that the output of the message is primarily meant for a human to read. You should avoid signaling a condition or the result of your program to another program by creating a message.

How should errors be handled?

Imagine writing a program that will take a long time to complete because of a complex calculation or because you're handling a large amount of data. If an error occurs during this computation then you're liable to lose all of the results that were calculated before the error, or your program may not finish a critical task that a program further down your pipeline is depending on. If you anticipate the possibility of errors occurring during the execution of your program then you can design your program to handle them appropriately.

The `tryCatch()` function is the workhorse of handling errors and warnings in R. The first argument of this function is any R expression, followed by conditions which specify how to handle an error or a warning. The last argument, `finally`, specifies a function or expression that will be executed after the expression no matter what, even in the event of an error or a warning.

Let's construct a simple function I'm going to call `beera` that catches errors and warnings gracefully.

```
beera <- function(expr){
  tryCatch(expr,
         error = function(e){
           message("An error occurred:\n", e)
         },
         warning = function(w){
           message("A warning occured:\n", w)
         },
         finally = {
           message("Finally done!")
         })
}
```

This function takes an expression as an argument and tries to evaluate it. If the expression can be evaluated without any errors or warnings then the result of the expression is returned and the message `Finally done!` is printed to the R console. If an error or warning is generated then the functions that are provided to the `error` or `warning` arguments are printed. Let's try this function out with a few examples.

```
beera({
  2 + 2
})
Finally done!
[1] 4

beera({
  "two" + 2
})
An error occurred:
Error in "two" + 2: non-numeric argument to binary operator
```

```
Finally done!

beera({
  as.numeric(c(1, "two", 3))
})
A warning occured:
simpleWarning in doTryCatch(return(expr), name, parentenv, handler): NAs introduced by coercion

Finally done!
```

Notice that we've effectively transformed errors and warnings into messages.

Now that you know the basics of generating and catching errors you'll need to decide when your program should generate an error. My advice to you is to limit the number of errors your program generates as much as possible. Even if you design your program so that it's able to catch and handle errors, the error handling process slows down your program by orders of magnitude. Imagine you wanted to write a simple function that checks if an argument is an even number. You might write the following:

```
is_even <- function(n){
  n %% 2 == 0
}

is_even(768)
[1] TRUE

is_even("two")
Error in n%%2: non-numeric argument to binary operator
```

You can see that providing a string causes this function to raise an error. You could imagine though that you want to use this function across a list of different data types, and you only want to know which elements of that list are even numbers. You might think to write the following:

```
is_even_error <- function(n){
  tryCatch(n %% 2 == 0,
           error = function(e){
             FALSE
           })
}

is_even_error(714)
[1] TRUE

is_even_error("eight")
[1] FALSE
```

This appears to be working the way you intended, however when applied to more data this function will be seriously slow compared to alternatives. For example I could check that `n` is numeric before treating `n` like a number:

```
is_even_check <- function(n){
  is.numeric(n) && n %% 2 == 0
}

is_even_check(1876)
[1] TRUE

is_even_check("twelve")
[1] FALSE
```

Notice that by using `is.numeric()` before the "AND" operator (`&&`) the expression `n %% 2 == 0` is never evaluated. This is a programming language design feature called "short circuiting." The expression can never evaluate to `TRUE` if the left hand side of `&&` evaluates to `FALSE`, so the right hand side is ignored.

To demonstrate the difference in the speed of the code we'll use the `microbenchmark` package to measure how long it takes for each function to be applied to the same data.

```
library(microbenchmark)
microbenchmark(sapply(letters, is_even_check))


Unit: microseconds
                           expr    min      lq     mean  median      uq     max neval
 sapply(letters, is_even_check) 46.224 47.7975 61.43616 48.6445 58.4755 167.091   100


microbenchmark(sapply(letters, is_even_error))


Unit: microseconds
                           expr     min       lq     mean   median       uq      max neval
 sapply(letters, is_even_error) 640.067 678.0285 906.3037 784.4315 1044.501 2308.931   100
```

The error catching approach is nearly 15 times slower!

Proper error handling is an essential tool for any software developer so that you can design programs that are error tolerant. Creating clear and informative error messages is essential for building quality software. One closing tip I recommend is to put documentation for your software online, including the meaning of the errors that your software can potentially throw. Often a user's first instinct when encountering an error is to search online for that error message, which should lead them to your documentation!

Summary

- Errors, warnings, and messages can be generated within R code using the functions `stop`, `stopifnot`, `warning`, and `message`.
- Catching errors, and providing useful error messaging, can improve user experience with functions but can also slow down code substantially.

2.6 Debugging

The learning objectives of this section are:

- Apply debugging tools to identify bugs in R programs

Debugging is the process of getting your expectations to converge with reality. When writing software in any language, we develop a certain set of expectations about how the software should behave and what it should do. But inevitably, when we run the software, it does something *different* from what we expected. In these situations, we need to engage in a process to determine if

1. Our expectations were incorrect, based on the documented behavior of the software; or
2. There is a problem with the code, such that the programming is not done in a way that will match expectations.

This is the process of debugging.

In the previous section, we discussed what to do when software generates conditions (errors, warnings, messages) in a manner that is completely *expected*. In those cases, we know that certain functions will generate errors and we want to handle them in a manner that is not the usual way.

This section describes the tools for debugging your software in R. R comes with a set of built-in tools for interactive debugging that can be useful for tracking down the source of problems. These functions are

- `browser()`: an interactive debugging environment that allows you to step through code one expression at a time
- `debug()` / `debugonce()`: a function that initiates the browser within a function
- `trace()`: a function that allows you to temporarily insert pieces of code into other functions to modify their behavior
- `recover()`: a function for navigating the function call stack after a function has thrown an error
- `traceback()`: a function that prints out the function call stack after an error occurs but does nothing if there's no error

`traceback()`

If an error occurs, the easiest thing to do is to immediately call the `traceback()` function. This function returns the function call stack just before the error occurred so that you can see what level of function calls the error occurred. If you have many functions calling each other in succession, the `traceback()` output can be useful for identifying where to go digging first.

For example, the following code gives an error.

```
check_n_value <- function(n) {
        if(n > 0) {
                stop("n should be <= 0")
        }
}
error_if_n_is_greater_than_zero <- function(n){
        check_n_value(n)
        n
}
error_if_n_is_greater_than_zero(5)
Error in check_n_value(n): n should be <= 0
```

Running the `traceback()` function immediately after getting this error would give us

```
traceback()
3: stop("n should be <= 0") at #2
2: check_n_value(n) at #2
1: error_if_n_is_greater_than_zero(5)
```

From the traceback, we can see that the error occurred in the `check_n_value()` function. Put another way, the `stop()` function was called from within the `check_n_value()` function.

Browsing a Function Environment

From the traceback output, it is often possible to determine in which function and on which line of code an error occurs. If you are the author of the code in question, one easy thing to do is to insert a call to the `browser()` function in the vicinity of the error (ideally, *before* the error occurs). The `browser()` function takes no arguments and is just placed wherever you want in the function. Once it is called, you will be in the browser environment, which is much like the regular R workspace environment except that you are inside a function.

```
check_n_value <- function(n) {
        if(n > 0) {
                browser()  ## Error occurs around here
                stop("n should be <= 0")
        }
}
```

Now, when we call `error_if_n_is_greater_than_zero(5)`, we will see the following.

```
error_if_n_is_greater_than_zero(5)
Called from: check_n_value(n)
Browse[1]>
```

Tracing Functions

If you have easy access to the source code of a function (and can modify the code), then it's usually easiest to insert `browser()` calls directly into the code as you track down various bugs. However, if you do not have easy access to a function's code, or perhaps a function is inside a package that would require rebuilding after each edit, it is sometimes easier to make use of the `trace()` function to make temporary code modifications.

The simplest use of `trace()` is to just call `trace()` on a function without any other arguments.

```
trace("check_n_value")
Error in trace("check_n_value"): could not find function "check_n_value"
```

Now, whenever `check_n_value()` is called by any other functions, you will see a message printed to the console indicating that the function was called.

```
error_if_n_is_greater_than_zero(5)
Error in check_n_value(n): n should be <= 0
```

Here we can see that `check_n_value()` was called once before the error occurred. But we can do more with `trace()`, such as inserting a call to `browser()` in a specific place, such as right before the call to `stop()`.

We can obtain the expression numbers of each part of a function by calling `as.list()` on the `body()` of a function.

```
as.list(body(check_n_value))
[[1]]
`{`

[[2]]
if (n > 0) {
    stop("n should be <= 0")
}
```

Here, the `if` statement is the second expression in the function (the first "expression" being the very beginning of the function). We can further break down the second expression as follows.

```
as.list(body(check_n_value)[[2]])
[[1]]
`if`

[[2]]
n > 0

[[3]]
{
    stop("n should be <= 0")
}
```

Now we can see the call to `stop()` is the third sub-expression within the second expression of the overall function. We can specify this to `trace()` by passing an integer vector wrapped in a list to the `at` argument.

```
trace("check_n_value", browser, at = list(c(2, 3)))
Error in getFunction(what, where = whereF): no function 'check_n_value' found
```

The `trace()` function has a side effect of modifying the function and converting into a new object of class "function".

```
check_n_value
function(n) {
        if(n > 0) {
                stop("n should be <= 0")
        }
}
<environment: 0x7fdc8e02e438>
```

You can see the internally modified code by calling

```
body(check_n_value)
{
    if (n > 0) {
        stop("n should be <= 0")
    }
}
```

Here we can see that the code has been altered to add a call to `browser()` just before the call to `stop()`.

We can add more complex expressions to a function by wrapping them in a call to `quote()` within the the `trace()` function. For example, we may only want to invoke certain behaviors depending on the local conditions of the function.

```
trace("check_n_value", quote({
        if(n == 5) {
                message("invoking the browser")
                browser()
        }
}), at = 2)
Error in getFunction(what, where = whereF): no function 'check_n_value' found
```

Here, we only invoke the `browser()` if `n` is specifically 5.

```
body(check_n_value)
{
    if (n > 0) {
        stop("n should be <= 0")
    }
}
```

Debugging functions within a package is another key use case for `trace()`. For example, if we wanted to insert tracing code into the `glm()` function within the `stats` package, the only addition to the `trace()` call we would need is to provide the namespace information via the `where` argument.

```
trace("glm", browser, at = 4, where = asNamespace("stats"))
Tracing function "glm" in package "namespace:stats"
[1] "glm"
```

Here we show the first few expressions of the modified `glm()` function.

```
body(stats::glm)[1:5]
{
    call <- match.call()
    if (is.character(family))
        family <- get(family, mode = "function", envir = parent.frame())
    {
        .doTrace(browser(), "step 4")
        if (is.function(family))
            family <- family()
    }
    if (is.null(family$family)) {
        print(family)
        stop("'family' not recognized")
    }
}
```

Using `debug()` and `debugonce()`

The `debug()` and `debugonce()` functions can be called on other functions to turn on the “debugging state” of a function. Calling `debug()` on a function makes it such that when that function is called, you immediately enter a browser and can step through the code one expression at a time.

```
## Turn on debugging state for 'lm' function
debug(lm)
```

A call to `debug(f)` where `f` is a function is basically equivalent to `trace(f, browser)` which will call the `browser()` function upon entering the function.

The debugging state is persistent, so once a function is flagged for debugging, it will remain flagged. Because it is easy to forget about the debugging state of a function, the `debugonce()` function turns on the debugging state the next time the function is called, but then turns it off after the browser is exited.

recover()

The `recover()` function is not often used but can be an essential tool when debugging complex code. Typically, you do not call `recover()` directly, but rather set it as the function to invoke anytime an error occurs in code. This can be done via the `options()` function.

```
options(error = recover)
```

Usually, when an error occurs in code, the code stops execution and you are brought back to the usual R console prompt. However, when `recover()` is in use and an error occurs, you are given the function call stack and a menu.

```
error_if_n_is_greater_than_zero(5)
Error in check_n_value(n) : n should be <= 0

Enter a frame number, or 0 to exit

1: error_if_n_is_greater_than_zero(5)
2: #2: check_n_value(n)

Selection:
```

Selecting a number from this menu will bring you into that function on the call stack and you will be placed in a browser environment. You can exit the browser and then return to this menu to jump to another function in the call stack.

The `recover()` function is very useful if an error is deep inside a nested series of function calls and it is difficult to pinpoint exactly where an error is occurring (so that you might use `browser()` or `trace()`). In such cases, the `debug()` function is often of little practical use because you may need to step through many many expressions before the error actually occurs. Another scenario is when there is a stochastic element to your code so that errors occur in an unpredictable way. Using `recover()` will allow you to browse the function environment only when the error eventually does occur.

Final Thoughts on Debugging

The debugging tools in any programming language can be essential for tracking down problems in code, especially when the code becomes complex and spans many lines. However, one should not lean on them too heavily so that they become a regular part of the programming process. It is easy to get into a situation where you "throw some code out there" and then let the debugger catch it before something bad happens. If you find yourself coding up a function and then immediately calling `debug()` on it, you are in this situation.

A better approach is to think carefully about what a function should do and then consider how to code it up. A few minutes of careful forethought can often save the hapless programmer hours of debugging.

Summary

- Debugging in R is facilitated with the functions `browser`, `debug`, `trace`, `recover`, and `traceback`.
- These debugging tools should not be used as a crutch when developing functions.

2.7 Profiling and Benchmarking

The learning objectives of this section are:

- Apply profiling and timing tools to optimize R code

Some of the R code that you write will be slow. Slow code often isn't worth fixing in a script that you will only evaluate a few times, as the time it will take to optimize the code will probably exceed the time it takes the computer to run it. However, if you are writing functions that will be used repeatedly, it is often worthwhile to identify slow sections of the code so you can try to improve speed in those sections.

In this section, we will introduce the basics of profiling R code, using functions from two packages, `microbenchmark` and `profvis`. The `profvis` package is fairly new and requires recent versions of both R (version 3.0 or higher) and RStudio. If you are having problems running either package, you should try updating both R and RStudio (the Preview version of RStudio, which will provide full functionality for `profvis`, is available for download here).

`microbenchmark`

The `microbenchmark` package is useful for running small sections of code to assess performance, as well as for comparing the speed of several functions that do the same thing. The `microbenchmark` function from this package will run code multiple times (100 times is the default) and provide summary statistics describing how long the code took to run across those iterations. The process of timing a function takes a certain amount of time itself.

The `microbenchmark` function adjusts for this overhead time by running a certain number of "warm-up" iterations before running the iterations used to time the code.

You can use the `times` argument in `microbenchmark` to customize how many iterations are used. For example, if you are working with a function that is a bit slow, you might want to run the code fewer times when benchmarking (although with slower or more complex code, it likely will make more sense to use a different tool for profiling, like `profvis`).

You can include multiple lines of code within a single call to `microbenchmark`. However, to get separate benchmarks of line of code, you must separate each line by a comma:

```
library(microbenchmark)
microbenchmark(a <- rnorm(1000),
               b <- mean(rnorm(1000)))
Unit: microseconds
                   expr    min     lq     mean  median      uq     max
       a <- rnorm(1000) 76.193 77.006 78.07966 77.4650 77.8815  93.349
 b <- mean(rnorm(1000)) 82.937 84.010 85.09241 84.4285 84.9290 100.769
 neval
   100
   100
```

The `microbenchmark` function is particularly useful for comparing functions that take the same inputs and return the same outputs. As an example, say we need a function that can identify days that meet two conditions: (1) the temperature equals or exceeds a threshold temperature (27 degrees Celsius in the examples) and (2) the temperature equals or exceeds the hottest temperature in the data before that day. We are aiming for a function that can input a data frame that includes a column named `temp` with daily mean temperature in Celsius, like this data frame:

date	temp
2015-07-01	26.5
2015-07-02	27.2
2015-07-03	28.0
2015-07-04	26.9
2015-07-05	27.5
2015-07-06	25.9
2015-07-07	28.0
2015-07-08	28.2

and outputs a data frame that has an additional binary `record_temp` column, specifying if that day meet the two conditions, like this:

```
date         temp   record_temp
2015-07-01   26.5   FALSE
2015-07-02   27.2   TRUE
2015-07-03   28.0   TRUE
2015-07-04   26.9   FALSE
2015-07-05   27.5   FALSE
2015-07-06   25.9   FALSE
2015-07-07   28.0   TRUE
2015-07-08   28.2   TRUE
```

Below are two example functions that can perform these actions. Since the `record_temp` column depends on temperatures up to that day, one option is to use a loop to create this value. The first function takes this approach. The second function instead uses tidyverse functions to perform the same tasks.

```r
# Function that uses a loop
find_records_1 <- function(datafr, threshold){
  highest_temp <- c()
  record_temp <- c()
  for(i in 1:nrow(datafr)){
    highest_temp <- max(highest_temp, datafr$temp[i])
    record_temp[i] <- datafr$temp[i] >= threshold &
      datafr$temp[i] >= highest_temp
  }
  datafr <- cbind(datafr, record_temp)
  return(datafr)
}

# Function that uses tidyverse functions
find_records_2 <- function(datafr, threshold){
  datafr <- datafr %>%
    mutate_(over_threshold = ~ temp >= threshold,
            cummax_temp = ~ temp == cummax(temp),
            record_temp = ~ over_threshold & cummax_temp) %>%
    select_(.dots = c("-over_threshold", "-cummax_temp"))
  return(as.data.frame(datafr))
}
```

If you apply the two functions to the small example data set, you can see that they both create the desired output:

```
example_data <- data_frame(date = c("2015-07-01", "2015-07-02",
                                    "2015-07-03", "2015-07-04",
                                    "2015-07-05", "2015-07-06",
                                    "2015-07-07", "2015-07-08"),
                           temp = c(26.5, 27.2, 28.0, 26.9,
                                    27.5, 25.9, 28.0, 28.2))

(test_1 <- find_records_1(example_data, 27))
        date temp record_temp
1 2015-07-01 26.5       FALSE
2 2015-07-02 27.2        TRUE
3 2015-07-03 28.0        TRUE
4 2015-07-04 26.9       FALSE
5 2015-07-05 27.5       FALSE
6 2015-07-06 25.9       FALSE
7 2015-07-07 28.0        TRUE
8 2015-07-08 28.2        TRUE

(test_2 <- find_records_2(example_data, 27))
        date temp record_temp
1 2015-07-01 26.5       FALSE
2 2015-07-02 27.2        TRUE
3 2015-07-03 28.0        TRUE
4 2015-07-04 26.9       FALSE
5 2015-07-05 27.5       FALSE
6 2015-07-06 25.9       FALSE
7 2015-07-07 28.0        TRUE
8 2015-07-08 28.2        TRUE

all.equal(test_1, test_2)
[1] TRUE
```

The performance of these two functions can be compared using `microbenchmark`:

```
record_temp_perf <- microbenchmark(find_records_1(example_data, 27),
                                   find_records_2(example_data, 27))
record_temp_perf
Unit: microseconds
                             expr      min       lq      mean    median
 find_records_1(example_data, 27)  700.617  739.050  800.1232  766.0245
 find_records_2(example_data, 27) 1057.485 1108.835 1265.6457 1143.6900
        uq      max neval
  814.4315 1102.318   100
 1220.2965 4735.997   100
```

This output gives summary statistics (`min`, `lq`, `mean`, `median`, `uq`, and `max`) describing the time it took to run the two function over the 100 iterations of each function call. By default, these times are given in a reasonable unit, based on the observed profiling times (units are given in microseconds in this case).

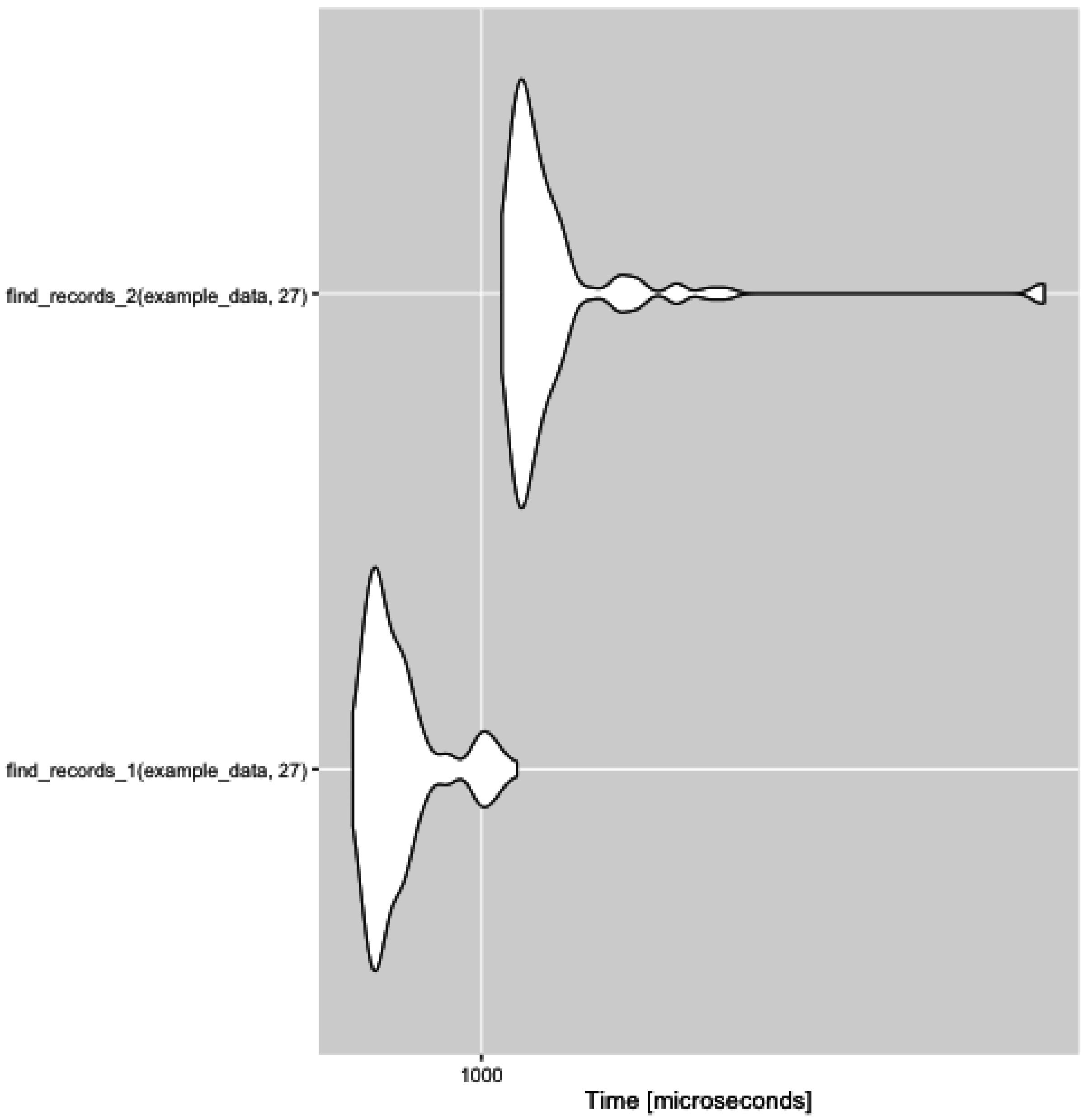
It's useful to check next to see if the relative performance of the two functions is similar for a bigger data set. The `chicagoNMMAPS` data set from the `dlnm` package includes temperature data over 15 years in Chicago, IL. Here are the results when we benchmark the two functions with that data (note, this code takes a minute or two to run):

```
library(dlnm)
data("chicagoNMMAPS")

record_temp_perf_2 <- microbenchmark(find_records_1(chicagoNMMAPS, 27),
                                     find_records_2(chicagoNMMAPS, 27))
record_temp_perf_2
Unit: milliseconds
                              expr        min         lq       mean
 find_records_1(chicagoNMMAPS, 27) 187.267026 200.899557 209.391010
 find_records_2(chicagoNMMAPS, 27)   1.857145   2.003118   2.372841
     median         uq       max neval
 206.678710 215.576020 315.67619   100
   2.222288   2.313402  11.77939   100
```

While the function with the loop (`find_records_1`) performed better with the very small sample data, the function that uses tidyverse functions (`find_records_2`) performs much, much better with a larger data set.

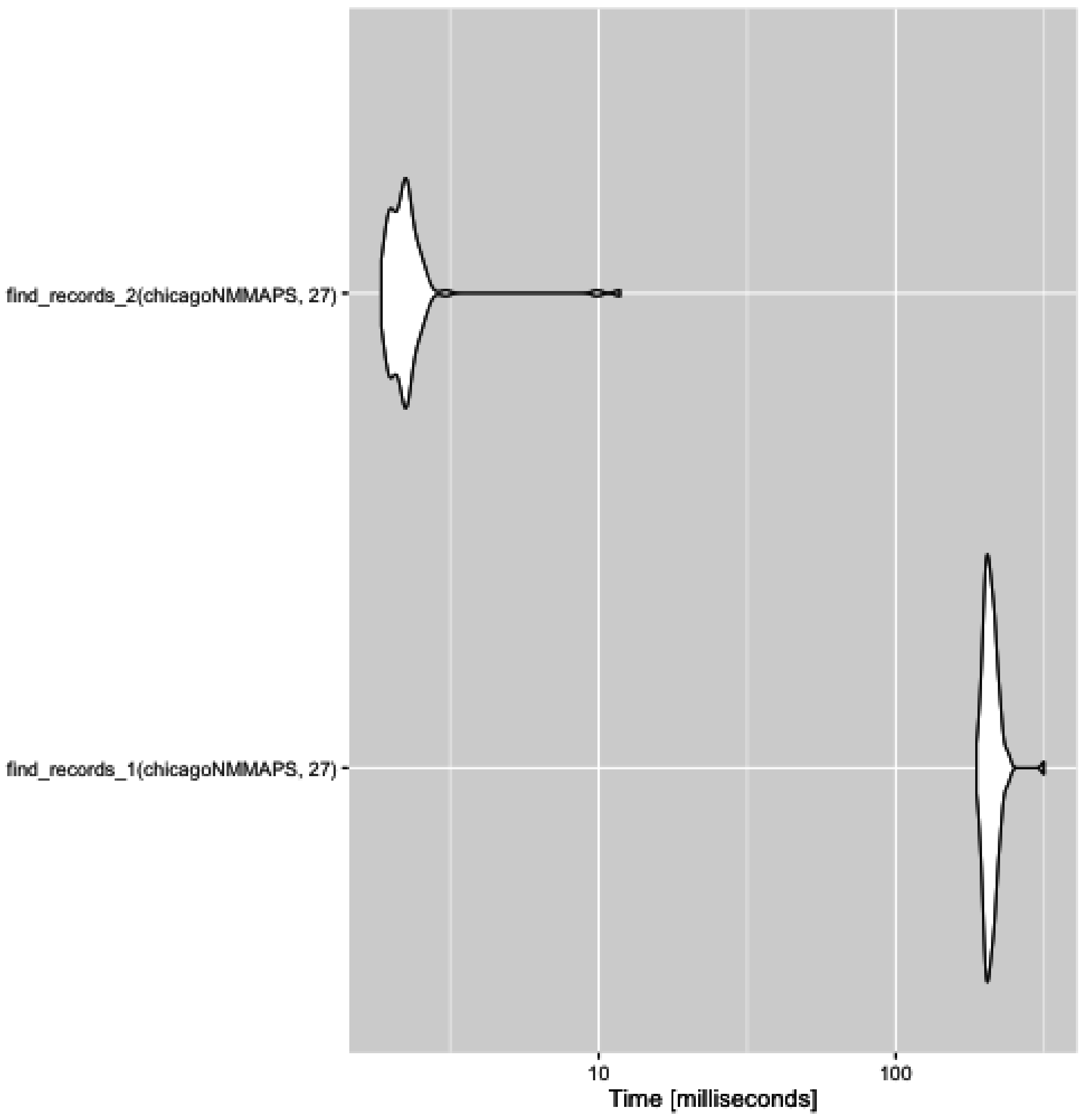
The `microbenchmark` function returns an object of the "microbenchmark" class. This class has two methods for plotting results, `autoplot.microbenchmark` and `boxplot.microbenchmark`. To use the `autoplot` method, you will need to have `ggplot2` loaded in your R session.

```
library(ggplot2)
# For small example data
autoplot(record_temp_perf)
```



Timing comparison of find records functions

```
# For larger data set
autoplot(record_temp_perf_2)
```



Timing comparison of find records functions

By default, this plot gives the “Time” axis on a log scale. You can change this with the argument `log = FALSE`.

`profvis`

Once you’ve identified slower code, you’ll likely want to figure out which parts of the code are causing bottlenecks. The `profvis` function from the `profvis` package is very useful for this type of profiling. This function uses the `RProf` function from base R to profile code, and then displays it in an interactive visualization in RStudio. This profiling is done by sampling, with the `RProf` function writing out the call stack every 10 milliseconds while running the code.

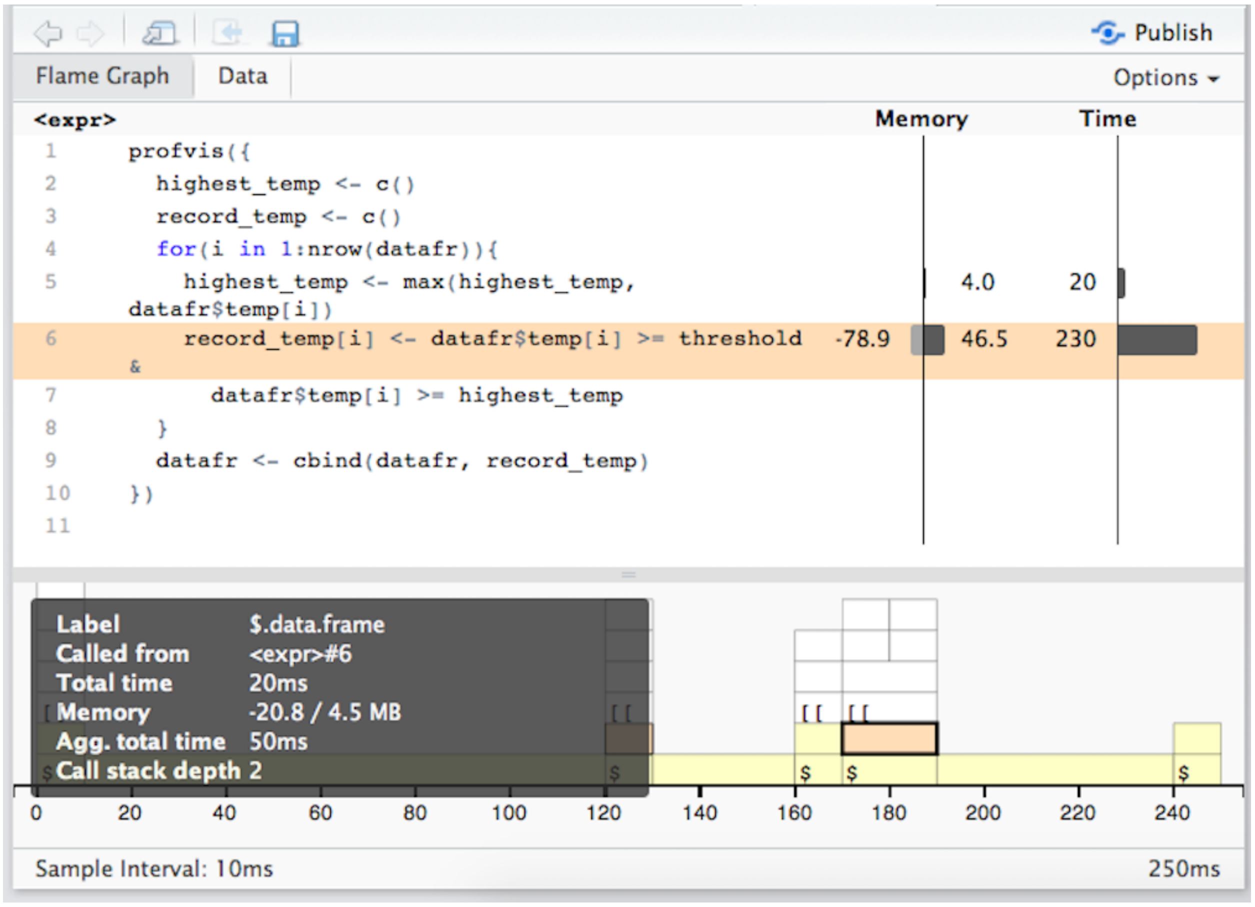
To profile code with `profvis`, just input the code (in braces if it is mutli-line) into `profvis` within RStudio. For example, we found that the `find_records_1` function was slow when used with a large data set. To profile the code in that function, run:

```
library(profvis)
datafr <- chicagoNMMAPS
threshold <- 27

profvis({
  highest_temp <- c()
  record_temp <- c()
  for(i in 1:nrow(datafr)){
    highest_temp <- max(highest_temp, datafr$temp[i])
    record_temp[i] <- datafr$temp[i] >= threshold &
      datafr$temp[i] >= highest_temp
  }
  datafr <- cbind(datafr, record_temp)
})
```

The `profvis` output gives you two options for visualization: “Flame Graph” or “Data” (a button to toggle between the two is given in the top left of the `profvis` visualization created when you profile code). The “Data” output defaults to show you the time usage of each first-level function call. Each of these calls can be expanded to show deeper and deeper functions calls within the call stack. This expandable interface allows you to dig down within a call stack to determine what calls are causing big bottlenecks. For functions that are part of a package you have loaded with `devtools::load_all`, this output includes a column with the file name where a given function is defined. This functionality makes this “Data” output pane particularly useful in profiling functions in a package you are creating.

The “Flame Graph” view in `profvis` output gives you two panels. The top panel shows the code called, with bars on the right to show memory use and time spent on the line. The bottom panel also visualizes the time used by each line of code, but in this case it shows time use horizontally and shows the full call stack at each time sample, with initial calls shown at the bottom of the graph, and calls deeper in the call stack higher in the graph. Clicking on a block in the bottom panel will show more information about a call, including which file it was called from, how much time it took, how much memory it took, and its depth in the call stack.

Figure @ref(fig:profvisexample) shows example output from profiling the code in the `find_-records_1` function defined earlier in this section.



```
profvis({
  highest_temp <- c()
  record_temp <- c()
  for(i in 1:nrow(datafr)){
    highest_temp <- max(highest_temp,
datafr$temp[i])
    record_temp[i] <- datafr$temp[i] >= threshold
&
      datafr$temp[i] >= highest_temp
  }
  datafr <- cbind(datafr, record_temp)
})
```

```
4.0   20
-78.9   46.5   230
```

```
Label            $.data.frame
Called from      <expr>#6
Total time       20ms
Memory           -20.8 / 4.5 MB
Agg. total time  50ms
Call stack depth 2
```

Example of profvis output for a function to find record temperatures.

Based on this visualization, most of the time is spent on line 6, filling in the `record_temp` vector. Now that we know this, we could try to improve the function, for example by doing a better job of initializing vectors before running the loop.

The `profvis` visualization can be used to profile code in functions you're writing as part of a package. If some of the functions in the code you are profiling are in a package currently loaded with loaded with `devtools::load_all`, the top panel in the Flame Graph output will include the code defining those functions, which allows you to explore speed and memory use within the code for each function. You can also profile code within functions from other packages– for more details on the proper set-up, see the "FAQ" section of RStudio's `profvis` documentation.

The `profvis` function will not be able to profile code that runs to quickly. Trying to profile functions that are too fast will give you the following error message:

```
Error in parse_rprof(prof_output, expr_source) :
  No parsing data available. Maybe your function was too fast?
```

You can use the argument `interval` in `profvis` to customize the sampling interval. The default is to sample every 10 milliseconds (`interval = 0.01`), but you can decrease this sampling interval. In some cases, you may be able to use this option to profile faster-running code. However, you should avoid using an interval smaller than about 5 milliseconds, as below that you will get inaccurate estimates with `profvis`. If you are running very fast code, you're better off profiling with `microbenchmark`, which can give accurate estimates at finer time intervals.

Here are some tips for optimizing your use of `profvis`:

- You may find it convenient to use the "Show in new window" button on the RStudio pane with profiling results to expand this window while you are interpreting results.
- An "Options" button near the top right gives different options for how to display the profiling results, including whether to include memory profiling results and whether to include lines of code with zero time.
- You can click-and-drag results in the bottom visualization panel, as well as pan in and out.
- You may need to update your version of RStudio to be able to use the full functionality of `profvis`. You can download a Preview version of RStudio here.
- If you'd like to share code profiling results from `profvis` publicly, you can do that by using the "Publish" button on the top right of the rendered profile visualization to publish the visualization to RPubs. The "FAQ" section of RStudio's `profvis` documentation includes more tips for sharing a code profile visualization online.
- If you get a lot of blocks labeled "<Anonymous>", try updating your version of R. In newer versions of R, functions called using `package::function()` syntax or `list$function()` syntax are labeled in profiling blocks in a more meaningful way. This is likely to be a particular concern if you are profiling code in a package you are developing, as you will often be using `package::function()` syntax extensively to pass CRAN checks.

Find out more

If you'd like to learn more about profiling R code, or improving performance of R code once you've profiled, you might find these resources helpful:

- RStudio's `profvis` documentation
- Section on performant code in Hadley Wickham's *Advanced R* book
- "FasteR! HigheR! StrongeR! - A Guide to Speeding Up R Code for Busy People", an article by Noam Ross

Summary

- Profiling can help you identify bottlenecks in R code.
- The `microbenchmark` package helps you profile short pieces of code and compare functions with each other. It runs the code many times and provides summary statistics across the iterations.
- The `profvis` package allows you to visualize performance across more extensive code. It can be used to profile code within functions being developed for a package, as long as the package source code has been loaded locally using `devtools::load_all`.

2.8 Non-standard evaluation

Functions from packages like `dplyr`, `tidyr`, and `ggplot2` are excellent for creating efficient and easy-to-read code that cleans and displays data. However, they allow shortcuts in calling columns in data frames that allow some room for ambiguity when you move from evaluating code interactively to writing functions for others to use. The non-standard evaluation used within these functions mean that, if you use them as you would in an interactive session, you'll get a lot of "no visible bindings" warnings when you run CRAN checks on your package. These warnings will look something like this:

```
map_counties: no visible binding for global variable 'fips'
map_counties: no visible binding for global variable 'storm_dist'
map_counties: no visible binding for global variable 'tot_precip'
Undefined global functions or variables:
  fips storm_dist tot_precip
```

When you write a function for others to use, you need to avoid non-standard evaluation and so avoid all of these functions (culprits include many `dplyr` and `tidyr` functions– including `mutate`, `select`, `filter`, `group_by`, `summarize`, `gather`, `spread`– but also some functions in `ggplot2`, including `aes`). Fortunately, these functions all have standard evaluation alternatives, which typically have the same function name followed by an underscore (for example, the standard evaluation version of `mutate` is `mutate_`).

The input to the function call will need to be a bit different for standard evaluation versions of these functions. In many cases, this change is as easy as using formula notation (~) within the call, but in some cases it requires something more complex, including using the `.dots` argument.

Here is a table with examples of non-standard evaluation calls and their standard evaluation alternatives (these are all written assuming that the function is being used as a step in a piping flow, where the input data frame has already been defined earlier in the piping sequence):

Non-standard evaluation version	Standard evaluation version
`filter(fips %in% counties)`	`filter_(~ fips %in% counties)`
`mutate(max_rain = max(tot_precip)`	`mutate_(max_rain = ~ max(tot_precip)`
`summarize(tot_precip = sum(precip))`	`summarize_(tot_precip = ~ sum(precip))`
`group_by(storm_id, fips)`	`group_by_(~ storm_id, ~ fips)`
`aes(x = long, y = lat)`	`aes_(x = ~ long, y = ~ lat)`
`select(-start_date, -end_date)`	`select_(.dots = c('start_date', 'end_date'))`
`select(-start_date, -end_date)`	`select_(.dots = c('-start_date', '-end_date'))`
`spread(key, mean)`	`spread_(key_col = 'key', value_col = 'mean')`
`gather(key, mean)`	`gather_(key_col = 'key', value_col = 'mean')`

If you have any non-standard evaluation in your package code (which you'll notice because of the "no visible bindings" warnings you'll get when you check the package), go through and change any instances to use standard evaluation alternatives. This change prevents these warnings when you check your package and will also ensure that the functions behave like you expect them to when they are run by other users in their own R sessions.

In this section, we've explained only how to convert from functions that use non-standard evaluation to those that use standard evaluation, to help in passing CRAN checks as you go from coding scripts to writing functions for packages. If you would like to learn more about non-standard evaluation in R, you should check out the chapter on non-standard evaluation in Hadley Wickham's *Advanced R* book.

Summary

- Functions that use non-standard evaluation can cause problems within functions written for a package.
- The NSE functions in tidyverse packages all have standard evaluation analogues that should be used when writing functions that will be used by others.

2.9 Object Oriented Programming

The learning objectives of this section are:

- Design and Implement a new S3, S4, or reference class with generics and methods

Introduction

Object oriented programming is one of the most successful and widespread philosophies of programming and is a cornerstone of many programming languages including Java, Ruby, Python, and C++. R has three object oriented systems because the roots of R date back to 1976, when the idea of object orientiented programming was barely four years old. New object oriented paradigms were added to R as they were invented, so some of the ideas in R about object oriented programming have gone stale in the years since. It's still important to understand these older systems since a huge amount of R code is written with them, and they're still useful and interesting! Long time object oriented programmers reading this book may find these old ideas refreshing.

The two older object oriented systems in R are called S3 and S4, and the modern system is called RC which stands for "reference classes." Programmers who are already familiar with object oriented programming will feel at home using RC.

Object Oriented Principles

There a several key principles in object oriented programming which span across R's object systems and other programming languages. The first are the ideas of a **class** and an **object**.

The world is made up of physical objects - the chair you're sitting in, the clock next to your bed, the bus you ride every day, etc. Just like the world is full of physical objects, your programs can be made of objects as well. A class is a blueprint for an object: it describes the parts of an object, how to make an object, and what the object is able to do. If you were to think about a class for a bus (as in the public buses that roam the roads) this class would describe attributes for the bus like the number of seats on the bus, the number of windows, the top speed of the bus, and the maximum distance the bus can drive on one tank of gas.

Buses in general can perform the same actions, and these actions are also described in the class: a bus can open and close its doors, the bus can steer, and the accelerator or the brake can be used to slow down or speed up the bus. Each of these actions can be described as a **method** which is a **function** that is associated with a particular class. We'll be using this class in order to create individual bus objects, so we should provide a **constructor** which is a method where we can specify attributes of the bus as arguments. This constructor method will then return an individual bus object with the attributes that we specified.

You could also imagine that after making the bus class you might want to make a special kind of class for a party bus. Party buses have all of the same attributes and methods as our bus class, but they also have additional attributes and methods like the number of refrigerators, window blinds that can be opened and closed, and smoke machines that can be turned on and off. Instead of rewriting the entire bus class and then adding new attributes and methods, it is possible for the party bus class to **inherit** all of the attributes and methods from the bus class. In this framework of inheritance, we talk about the bus class as the super-class of the party bus, and the party bus is the sub-class of the bus. What this relationship means is that the party bus has all of the same attributes and methods as the bus class plus additional attributes and methods.

S3

Conveniently everything in R is an object. By "everything" I mean every single "thing" in R including numbers, functions, strings, data frames, lists, etc. If you want to know the class of an object in R you can simply use the `class()` function:

```
class(2)
[1] "numeric"
class("is in session.")
[1] "character"
class(class)
[1] "function"
```

Now it's time to wade into some of the quirks of R's object oriented systems. In the S3 system you can arbitrarily assign a class to any object, which goes against most of what we discussed in the *Object Oriented Principles* section. Class assignments can be made using the `structure()` function, or you can assign the class using `class()` and `<-`:

```
special_num_1 <- structure(1, class = "special_number")
class(special_num_1)
[1] "special_number"

special_num_2 <- 2
class(special_num_2)
[1] "numeric"
class(special_num_2) <- "special_number"
class(special_num_2)
[1] "special_number"
```

This is completely legal R code, but if you want to have a better behaved S3 class you should create a constructor which returns an S3 object. The `shape_S3()` function below is a constructor that returns a shape_S3 object:

```
shape_s3 <- function(side_lengths){
  structure(list(side_lengths = side_lengths), class = "shape_S3")
}

square_4 <- shape_s3(c(4, 4, 4, 4))
class(square_4)
[1] "shape_S3"

triangle_3 <- shape_s3(c(3, 3, 3))
class(triangle_3)
[1] "shape_S3"
```

We've now made two shape_S3 objects: `square_4` and `triangle_3`, which are both instantiations of the shape_S3 class. Imagine that you wanted to create a method that would return `TRUE` if a shape_S3 object was a square, `FALSE` if a shape_S3 object was not a square, and `NA` if the object providied as an argument to the method was not a shape_s3 object. This can be achieved using R's **generic methods** system. A generic method can return different values based depending on the class of its input. For example `mean()` is a generic method that can find the average of a vector of number or it can find the "average day" from a vector of dates. The following snippet demonstrates this behavior:

```
mean(c(2, 3, 7))
[1] 4
mean(c(as.Date("2016-09-01"), as.Date("2016-09-03")))
[1] "2016-09-02"
```

Now let's create a generic method for identifying shape_S3 objects that are squares. The creation of every generic method uses the `UseMethod()` function in the following way with only slight variations:

```
[name of method] <- function(x) UseMethod("[name of method]")
```

Let's call this method `is_square`:

```
is_square <- function(x) UseMethod("is_square")
```

Now we can add the actual function definition for detecting whether or not a shape is a square by specifying `is_square.shape_S3`. By putting a dot (`.`) and then the name of the class after `is_squre`, we can create a method that associates `is_squre` with the `shape_S3` class:

```
is_square.shape_S3 <- function(x){
  length(x$side_lengths) == 4 &&
    x$side_lengths[1] == x$side_lengths[2] &&
    x$side_lengths[2] == x$side_lengths[3] &&
    x$side_lengths[3] == x$side_lengths[4]
}

is_square(square_4)
[1] TRUE
is_square(triangle_3)
[1] FALSE
```

Seems to be working well! We also want `is_square()` to return `NA` when its argument is not a shape_S3. We can specify `is_square.default` as a last resort if there is not method associated with the object passed to `is_square()`.

```
is_square.default <- function(x){
  NA
}

is_square("square")
[1] NA
is_square(c(1, 1, 1, 1))
[1] NA
```

Let's try printing `square_4`:

```
print(square_4)
$side_lengths
[1] 4 4 4 4

attr(,"class")
[1] "shape_S3"
```

Doesn't that look ugly? Lucky for us `print()` is a generic method, so we can specify a print method for the shape_S3 class:

```
print.shape_S3 <- function(x){
  if(length(x$side_lengths) == 3){
    paste("A triangle with side lengths of", x$side_lengths[1],
          x$side_lengths[2], "and", x$side_lengths[3])
  } else if(length(x$side_lengths) == 4) {
    if(is_square(x)){
      paste("A square with four sides of length", x$side_lengths[1])
    } else {
      paste("A quadrilateral with side lengths of", x$side_lengths[1],
            x$side_lengths[2], x$side_lengths[3], "and", x$side_lengths[4])
    }
  } else {
    paste("A shape with", length(x$side_lengths), "slides.")
  }
}

print(square_4)
[1] "A square with four sides of length 4"
print(triangle_3)
[1] "A triangle with side lengths of 3 3 and 3"
print(shape_s3(c(10, 10, 20, 20, 15)))
[1] "A shape with 5 slides."
print(shape_s3(c(2, 3, 4, 5)))
[1] "A quadrilateral with side lengths of 2 3 4 and 5"
```

Since printing an object to the console is one of the most common things to do in R, nearly every class has an assocaited print method! To see all of the methods associated with a generic like `print()` use the `methods()` function:

```
head(methods(print), 10)
 [1] "print,ANY-method"            "print,diagonalMatrix-method"
 [3] "print,sparseMatrix-method"   "print.acf"
 [5] "print.anova"                 "print.anova.gam"
 [7] "print.anova.lme"             "print.aov"
 [9] "print.aovlist"               "print.ar"
```

One last note on S3 with regard to inheritance. In the previous section we discussed how a sub-class can inhert attributes and methods from a super-class. Since you can assign any class to an object in S3, you can specify a super class for an object the same way you would specify a class for an object:

```
class(square_4)
[1] "shape_S3"
class(square_4) <- c("shape_S3", "square")
class(square_4)
[1] "shape_S3" "square"
```

To check if an object is a sub-class of a specified class you can use the `inherits()` function:

```
inherits(square_4, "square")
[1] TRUE
```

Example: S3 Class/Methods for Polygons

The S3 system doesn't have a formal way to define a class but typically, we use a list to define the class and elements of the list serve as data elements.

Here is our definition of a polygon represented using Cartesian coordinates. The class contains an element called `xcoord` and `ycoord` for the x- and y-coordinates, respectively. The `make_poly()` function is the "constructor" function for polygon objects. It takes as arguments a numeric vector of x-coordinates and a corresponding numeric vector of y-coordinates.

```
## Constructor function for polygon objects
## x a numeric vector of x coordinates
## y a numeric vector of y coordinates
make_poly <- function(x, y) {
        if(length(x) != length(y))
                stop("'x' and 'y' should be the same length")

        ## Create the "polygon" object
        object <- list(xcoord = x, ycoord = y)

        ## Set the class name
        class(object) <- "polygon"
        object
}
```

Now that we have a class definition, we can develop some methods for operating on objects from that class.

The first method we'll define is the `print()` method. The `print()` method should just show some simple information about the object and should not be too verbose—just enough information that the user knows what the object is.

Here the `print()` method just shows the user how many vertices the polygon has. It is a convention for `print()` methods to return the object `x` invisibly.

```
## Print method for polygon objects
## x an object of class "polygon"
print.polygon <- function(x, ...) {
        cat("a polygon with", length(x$xcoord),
            "vertices\n")
        invisible(x)
}
```

Next is the `summary()` method. The `summary()` method typically shows a bit more information and may even do some calculations. This `summary()` method computes the ranges of the x- and y-coordinates.

The typical approach for `summary()` methods is to allow the summary method to compute something, but to *not* print something. The strategy is

1. The `summary()` method returns an object of class "summary_'class name'"
2. There is a separate `print()` method for "summary_'class name'" objects.

For example, here is the `summary()` method.

```
## Summary method for polygon objects
## object an object of class "polygon"

summary.polygon <- function(object, ...) {
        object <- list(rng.x = range(object$xcoord),
                       rng.y = range(object$ycoord))
        class(object) <- "summary_polygon"
        object
}
```

Note that it simply returns an object of class `summary_polygon`. Now the corresponding `print()` method.

```
## Print method for summary.polygon objects
## x an object of class "summary_polygon"
print.summary_polygon <- function(x, ...) {
        cat("x:", x$rng.x[1], "-->", x$rng.x[2], "\n")
        cat("y:", x$rng.y[1], "-->", x$rng.y[2], "\n")
        invisible(x)
}
```

Now we can make use of our new class and methods.

```
## Construct a new "polygon" object
x <- make_poly(1:4, c(1, 5, 2, 1))
```

We can use the `print()` to see what the object is.

```
print(x)
a polygon with 4 vertices
```

And we can use the `summary()` method to get a bit more information about the object.

```
out <- summary(x)
class(out)
[1] "summary_polygon"
print(out)
x: 1 --> 4
y: 1 --> 5
```

Because of auto-printing we can just call the `summary()` method and let the results auto-print.

```
summary(x)
$rng.x
[1] 1 4

$rng.y
[1] 1 5

attr(,"class")
[1] "summary_polygon"
```

From here, we could build other methods for interacting with our `polygon` object. For example, it may make sense to define a `plot()` method or maybe methods for intersecting two polygons together.

S4

The S4 system is slightly more restrictive than S3, but it's similar in many ways. To create a new class in S4 you need to use the `setClass()` function. You need to specify two or three arguments for this function: `Class` which is the name of the class as a string, `slots`, which is a named list of attributes for the class with the class of those attributes specified, and optionally `contains` which includes the super-class of they class you're specifying (if there is a super-class). Take look at the class definition for a `bus_S4` and a `party_bus_S4` below:

```
setClass("bus_S4",
         slots = list(n_seats = "numeric",
                      top_speed = "numeric",
                      current_speed = "numeric",
                      brand = "character"))
setClass("party_bus_S4",
         slots = list(n_subwoofers = "numeric",
                      smoke_machine_on = "logical"),
         contains = "bus_S4")
```

Now that we've created the `bus_S4` and the `party_bus_S4` classes we can create bus objects using the `new()` function. The `new()` function's arguments are the name of the class and values for each "slot" in our S4 object.

```
my_bus <- new("bus_S4", n_seats = 20, top_speed = 80,
              current_speed = 0, brand = "Volvo")
my_bus
An object of class "bus_S4"
Slot "n_seats":
[1] 20

Slot "top_speed":
[1] 80

Slot "current_speed":
[1] 0

Slot "brand":
[1] "Volvo"
my_party_bus <- new("party_bus_S4", n_seats = 10, top_speed = 100,
                    current_speed = 0, brand = "Mercedes-Benz",
                    n_subwoofers = 2, smoke_machine_on = FALSE)
my_party_bus
An object of class "party_bus_S4"
Slot "n_subwoofers":
[1] 2

Slot "smoke_machine_on":
[1] FALSE

Slot "n_seats":
[1] 10

Slot "top_speed":
[1] 100

Slot "current_speed":
[1] 0

Slot "brand":
[1] "Mercedes-Benz"
```

You can use the `@` operator to access the slots of an S4 object:

```
my_bus@n_seats
[1] 20
my_party_bus@top_speed
[1] 100
```

This is essentially the same as using the `$` operator with a list or an environment.

S4 classes use a generic method system that is similar to S3 classes. In order to implement a new generic method you need to use the `setGeneric()` function and the `standardGeneric()` function in the following way:

```
setGeneric("new_generic", function(x){
  standardGeneric("new_generic")
})
```

Let's create a generic function called `is_bus_moving()` to see if a bus_S4 object is in motion:

```
setGeneric("is_bus_moving", function(x){
  standardGeneric("is_bus_moving")
})
[1] "is_bus_moving"
```

Now we need to actually define the function which we can to with `setMethod()`. The `setMethod()` functions takes as arguments the name of the method as a stirng, the method signature which specifies the class of each argument for the method, and then the function definition of the method:

```
setMethod("is_bus_moving",
          c(x = "bus_S4"),
          function(x){
            x@current_speed > 0
          })
[1] "is_bus_moving"

is_bus_moving(my_bus)
[1] FALSE
my_bus@current_speed <- 1
is_bus_moving(my_bus)
[1] TRUE
```

In addition to creating your own generic methods, you can also create a method for your new class from an existing generic. First use the `setGeneric()` function with the name of the existing method you want to use with your class, and then use the `setMethod()` function like in the previous example. Let's make a `print()` method for the bus_S4 class:

```
setGeneric("print")
[1] "print"

setMethod("print",
          c(x = "bus_S4"),
          function(x){
            paste("This", x@brand, "bus is traveling at a speed of", x@current_speed)
          })
[1] "print"

print(my_bus)
[1] "This Volvo bus is traveling at a speed of 1"
print(my_party_bus)
[1] "This Mercedes-Benz bus is traveling at a speed of 0"
```

Reference Classes

With reference classes we leave the world of R's old object oriented systems and enter the philosophies of other prominent object oriented programming languages. We can use the `setRefClass()` function to define a class' fields, methods, and super-classes. Let's make a reference class that represents a student:

```
Student <- setRefClass("Student",
                      fields = list(name = "character",
                                    grad_year = "numeric",
                                    credits = "numeric",
                                    id = "character",
                                    courses = "list"),
                      methods = list(
                        hello = function(){
                          paste("Hi! My name is", name)
                        },
                        add_credits = function(n){
                          credits <<- credits + n
                        },
                        get_email = function(){
                          paste0(id, "@jhu.edu")
                        }
                      ))
```

To recap: we've created a class definition called `Student` which defines the student class. This class has five fields and three methods. To create a Student object use the `new()` method:

```
brooke <- Student$new(name = "Brooke", grad_year = 2019, credits = 40,
                    id = "ba123", courses = list("Ecology", "Calculus III"))
roger <- Student$new(name = "Roger", grad_year = 2020, credits = 10,
                    id = "rp456", courses = list("Puppetry", "Elementary Algebra"))
```

You can access the fields and methods of each object using the `$` operator:

```
brooke$credits
[1] 40
roger$hello()
[1] "Hi! My name is Roger"
roger$get_email()
[1] "rp456@jhu.edu"
```

Methods can change the state of an object, for instance in the case of the `add_credits()` function:

```
brooke$credits
[1] 40
brooke$add_credits(4)
brooke$credits
[1] 44
```

Notice that the `add_credits()` method uses the complex assignment operator (`<<-`). You need to use this operator if you want to modify one of the fields of an object with a method. You'll learn more about this operator in the Expressions & Environments section.

Reference classes can inheret from other classes by specifying the `contains` argument when they're defined. Let's create a sub-class of Student called Grad_Student which includes a few extra features:

```
Grad_Student <- setRefClass("Grad_Student",
                            contains = "Student",
                            fields = list(thesis_topic = "character"),
                            methods = list(
                              defend = function(){
                                paste0(thesis_topic, ". QED.")
                              }
                            ))

jeff <- Grad_Student$new(name = "Jeff", grad_year = 2021, credits = 8,
                    id = "jl55", courses = list("Fitbit Repair",
                                                "Advanced Base Graphics"),
                    thesis_topic = "Batch Effects")

jeff$defend()
[1] "Batch Effects. QED."
```

Summary

- R has three object oriented systems: S3, S4, and Reference Classes.
- Reference Classes are the most similar to classes and objects in other programming languages.
- Classes are blueprints for an object.
- Objects are individual instances of a class.
- Methods are functions that are associaed with a particular class.
- Constructors are methods that create objects.
- Everything in R is an object.
- S3 is a liberal object oriented system that allows you to assign a class to any object.
- S4 is a more strict object oriented system that build upon ideas in S3.
- Reference Classes are a modern object oriented system that is similar to Java, C++, Python, or Ruby.

2.10 Gaining Your 'tidyverse' Citizenship

The learning objectives of this section are:

- Describe the principles of tidyverse functions

Many of the tools that we discuss in this book revolve around the so-called "tidyverse" set of tools. These tools, largely developed by Hadley Wickham but also including a diverse community of developers, have a set of principles that are adhered to when they are being developed. Hadley Wicham laid out these principles in his Tidy Tools Manifesto, a vignette within the `tidyverse` package.

The four basic principles of the tidyverse are:

Reuse existing data structures

R has a number of data structures (data frames, vectors, etc.) that people have grown accustomed to over the many years of R's existence. While it is often tempting to develop custom data structures, for example, by using S3 or S4 classes, it is often worthwhile to consider reusing a commonly used structure. You'll notice that many tidyverse functions make heavy use of the data frame (typically as their first argument), because the data frame is a well-known, well-understood structure used by many analysts. Data frames have a well-known and reasonably standardized corresponding file format in the CSV file.

While common data structures like the data frame may not be perfectly suited to your needs as you develop your own software, it is worth considering using them anyway because the enormous value to the community that is already familiar with them. If the user community feels familiar with the data structures required by your code, they are likely to adopt them quicker.

Compose simple functions with the pipe

One of the original principles of the Unix operating system was that every program should do "one thing well". The limitation of only doing one thing (but well!) was removed by being able to easily pipe the output of one function to be the input of another function (the pipe operator on Unix was the `|` symbol). Typical Unix commands would contain long strings commands piped together to (eventually) produce some useful output. On Unix systems, the unifying concept that allowed programs to pipe to each other was the use of [textual formats]. All data was rendered in textual formats so that if you wrote a new program, you would not need to worry about decoding some obscure proprietary format.

Much like the original Unix systems, the tidyverse eschews building monolithic functions that have many bells and whistles. Rather, once you are finished writing a simple function, it is better to start afresh and work off the input of another function to produce new output (using the `%>%` operator, for example). The key to this type of development is having *clean interfaces* between functions and an expectation that the output of every function may serve as the input to another function. This is why the first principle (reuse existing data structures) is important, because the reuse of data structures that are well-understood and characterized lessens the burden on other developers who are developing new code and would prefer not to worry about new-fangled data structures at every turn.

Embrace functional programming

This can be a tough principle for people coming from other non-functional programming languages. But the reality is, R is a functional programming language (with its roots in Scheme) and it's best not to go against the grain. In our section on Functional Programming, we outlined many of the principles that are fundamental to functional-style programming. In particular, the `purrr` package implements many of those ideas.

One benefit to functional programming is that it can at times be easier to reason about when simply looking at the code. The inability to modify arguments enables us to predict what the output of a function will be given a certain input, allowing for things like memoization. Functional programming also allows for simple parallelization, so that we can quickly parallelize any code that uses `lapply()` or `map()`.

Design for humans

Making your code *readable* and *usable* by people is goal that is overlooked surprisingly often. The result is things like function names that are obscure and do not actually communicate what they do. When writing code, using things like good, explicit, function names, with descriptive arguments, can allow for users to quickly learn your API. If you have a set of functions with a similar purpose, they might share a prefix (see e.g. `geom_point()`, `geom_line()`, etc.). If you have an argument like `color` that could either take arguments `1`, `2`, and `3`, or `black`, `red`, and `green`, think about which set of arguments might be easier for humans to handle.

About the Authors

Roger D. Peng is a Professor of Biostatistics at the Johns Hopkins Bloomberg School of Public Health. He is also a Co-Founder of the Johns Hopkins Data Science Specialization, which has enrolled over 1.5 million students, the Johns Hopkins Executive Data Science Specialization, the Simply Statistics blog where he writes about statistics and data science for the general public, and the Not So Standard Deviations podcast. Roger can be found on Twitter and GitHub under the user name rdpeng.

Sean Kross is a software developer in the Department of Biostatistics at the Johns Hopkins Bloomberg School of Public Health. Sean's professional interests range between metagenomics, cybersecurity, and human-computer interaction. He is also the lead developer of swirl, a software package designed to teach programming, statistics, and data science in an authentic programming environment. You can find Sean on Twitter and GitHub at seankross.

Brooke Anderson is an Assistant Professor at Colorado State University in the Department of Environmental & Radiological Health Sciences, as well as a Faculty Associate in the Department of Statistics. She is also a member of the universityâ€™s Partnership of Air Quality, Climate, and Health and is a member of the editorial boards of *Epidemiology* and *Environmental Health Perspectives*. Previously, she completed a postdoctoral appointment in Biostatistics at Johns Hopkins Bloomberg School of Public and a PhD in Engineering at Yale University. Her research focuses on the health risks associated with climate-related exposures, including heat waves and air pollution, for which she has conducted several national-level studies. As part of her research, she has also published a number of open source R software packages to facilitate environmental epidemiologic research.