DP4SS

Jesse Lecy Jamison Crawford

2024-06-12

Table of contents

Chapter 1

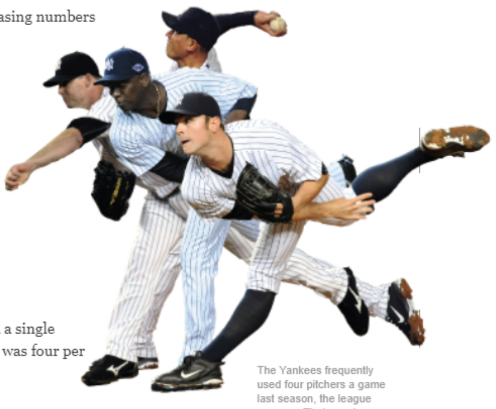
Introduction

Theories as to why hitters are striking out in ever-increasing numbers reflect changes in both pitching and hitting.

Fresh Arms Keep Coming

The days of a starter going deep into a game and then giving way to his closer appear long gone. Managers now have extensive scouting reports of the opponent's hitters against his pitchers. This has led to more specialization, more pitching changes — and more strikeouts. Hitters are now facing more fresh arms and ever-diminishing odds.

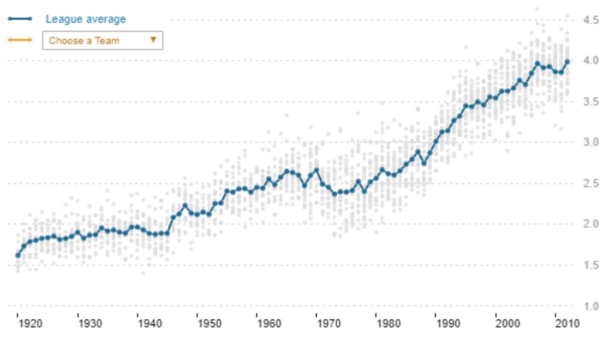
In 1924, the year with the fewest strikeouts, teams used a single pitcher nearly half of the time. Last season, the average was four per game.



The Yankees frequently used four pitchers a game last season, the league average. Their most common four-man combination was Freddy Garcia, Boone Logan, David Robertson and Rafael Soriano.

Barton Silverman/The New York Times (Soriano, Garcia); Richard Perry/The New York Times (Logan); Jason Szenes for The New York Times (Robertson)

Number of pitchers per team per game



Chapter 2

Introduction to R

This lecture introduces you to basic operations when you first start using R such as navigation, the object-oriented framework, loading a package, and creating some data vectors.

2.1 Navigation

You need to know a few operations to help you maneuver the R work environment, such as listing objects (datasets and functions) that are active, changing your working directory, listing available files, and finding help.

2.1.1 Setting Your Working Directory

When you are ready to load data, R needs to know where to look for your files. You can check what is avaiable in the current directory (i.e. folder) by asking to list all of the current files using **dir()**.

```r
dir()
```

If the file that you need is located in a different folder, you can change directories easily in R Studio by Session -> Set working director -> Choose directory (or Ctrl + Shift + H).

If you are writing a script, you want to keep track of this step so that it can be reproduced. Use the function **get.wd()** to check your current working directory, and **set.wd()** to change. You need to specify your path as an argument to this function, such as.

```r
setwd( "C:/user/projects/financial model" )
```

NOTE! R uses unix style notation with forward slashes, so if you copy and paste from Windows it will look like this, with back slashes:

```
setwd( "C:\user\projects\financial model" )
```

You will need to change them around for it to work.

It is best to save all of your steps in your scripts so that the analysis can be reproduced by yourself or others. In some cases you are doing exploratory or summary work, and you may want to find a file a quickly. You can use the **file.choose()** function to open a GUI to select your file directly. This function is used as an argument inside of a load data function.

```
my.dat <- read.csv( file.choose() )
```

2.2 Commenting Code

Most computer languages have a special character that is used to "comment out" lines so that it is not run by the program. It is used for two important purposes. First, we can add text to document our functions and it will not interfere with the program. And two, we can use it to run a program while ignoring some of the code, often for debugging purposes.

The # hash tag is used for comments in R.

```
##===============================================
##
##  Here is some documentation for this script
##
##===============================================

x <- 1:10

sum( x )
```

```
[1] 55
```

```
# y <- 1:25      # not run

# sum( y )       # not run
```

2.3 Help

You will use the help functions frequently to figure out what arguments and values are needed for specific functions. Because R is very customizable, you will find that many functions have several or dozens of arguments, and it is difficult to remember the correct syntax and values. But don't worry, to look them up all you need is the function name and a call for help:

help(dotchart) # opens an external helpfile

If you just need to remind yourself which arguments are defined in a function, you can use the *args()* command:

```
args( dotchart )
```

```
function (x, labels = NULL, groups = NULL, gdata = NULL, offset = 1/8,
    ann = par("ann"), xaxt = par("xaxt"), frame.plot = TRUE,
    log = "", cex = par("cex"), pt.cex = cex, pch = 21, gpch = 21,
    bg = par("bg"), color = par("fg"), gcolor = par("fg"), lcolor = "gray",
    xlim = range(x[is.finite(x)]), main = NULL, xlab = NULL,
    ylab = NULL, ...)
NULL
```

If you can't recall a function name, you can list all of the functions from a specific package as follows:

help(package="stats") # lists all functions in stats package

2.4 Install Programs (packages)

When you open R by default it will launch a core set of programs, called "packages" in R speak, that are use for most data operations. To see which packages are currently active use the **search()** function.

```
search()
```

```
[1] ".GlobalEnv"        "package:stats"     "package:graphics"
[4] "package:grDevices" "package:utils"     "package:datasets"
[7] "package:methods"   "Autoloads"         "package:base"
```

These programs manage the basic data operations, run the core graphics engine, and give you basic statistical methods.

The real magic for R comes from the over 7,000 contributed packages available on the CRAN: https://cran.r-project.org/web/views/

A package consists of custom functions and datasets that are generated by users. They are *packaged* together so that they can be shared with others. A package also includes documentation that describes each function, defines all of the arguments, and documents any datasets that are included.

If you know a package name, it is easy to install. In R Studio you can select Tools -> Install Packages and a list of available packages will be generated. But it is easier to use the **install.packages()** command. We will use the Lahman Package in this course, so let's install that now.

Description *This package provides the tables from Sean Lahman's Baseball Database as a set of R data.frames. It uses the data on pitching, hitting and fielding performance and other tables from 1871 through 2013, as recorded in the 2014 version of the database.*

See the documentation here: https://cran.r-project.org/web/packages/Lahman/Lahman.pdf

```
install.packages( "Lahman" )
```

You will be asked to select a "mirror". In R speak this just means the server from which you will download the package (choose anything nearby). R is a community of developers and universities that create code and maintain the infrastructure. A couple of dozen universities around the world host servers that contain copies of the R packages so that they can be easily accessed everywhere.

If the package is successfully installed you will get a message similar to this:

> package 'Lahman' successfully unpacked and MD5 sums checked

Once a new program is installed you can now open ("load" in R speak) the package using the **library()** command:

```
library( "Lahman" )
```

If you now type **search()** you can see that Lahman has been added to the list of active programs. We can now access all of the functions and data that are available in the Lahman package.

2.5 Accessing Built-In Datasets in R

One nice feature of R is that is comes with a bunch of built-in datasets that have been contributed by users are are loaded automatically. You can see the list of available datasets by typing:

```
data()
```

This will list all of the default datasets in core R packages. If you want to see all of the datasets available in installed packages as well use:

```
data( package = .packages(all.available = TRUE) )
```

2.5.1 Basic Data Operations

Let's ignore the underlying data structure right now and look at some ways that we might interact with data.

We will use the **USArrests** dataset available in the core files.

To access the data we need to load it into working memory. Anything that is active in R will be listed in the environment, which you can check using the **ls()** command. We will load the dataset using the **data()** command.

```
remove( list=ls() )
```

```
ls() # nothing currently available
```

```
character(0)
```

```r
data( "USArrests" )
ls() # data is now available for use
```

```
[1] "USArrests"
```

Now that we have loaded a dataset, we can start to access the variables and analyze relationships. Let's get to know our dataset.

```r
names( USArrests )  # which variables are in the dataset?
```

```
[1] "Murder"   "Assault"  "UrbanPop" "Rape"
```

```r
nrow( USArrests )   # how many observations are there?
```

```
[1] 50
```

```r
dim( USArrests )    # a quick way to see rows and columns
```

```
[1] 50  4
```

```r
# observation labels (row names) in our data:
row.names( USArrests ) |> head()
```

```
[1] "Alabama"    "Alaska"     "Arizona"    "Arkansas"   "California"
[6] "Colorado"
```

```r
# summary statistics for each variable
summary( USArrests )  |> pander::pander()
```

Murder	Assault	UrbanPop	Rape
Min. : 0.800	Min. : 45.0	Min. :32.00	Min. : 7.30
1st Qu.: 4.075	1st Qu.:109.0	1st Qu.:54.50	1st Qu.:15.07
Median : 7.250	Median :159.0	Median :66.00	Median :20.10
Mean : 7.788	Mean :170.8	Mean :65.54	Mean :21.23
3rd Qu.:11.250	3rd Qu.:249.0	3rd Qu.:77.75	3rd Qu.:26.18
Max. :17.400	Max. :337.0	Max. :91.00	Max. :46.00

We can see that the dataset consists of four variables: Murder, Assault, UrbanPop, and Rape. We also see that our unit of analysis is the state. But where does the data come from, and how are these variables measured?

To see the documentation for a specific dataset you will need to use the **help()** function:

```r
help( "USArrests" )
```

We get valuable information about the source and metrics:

Description *This data set contains statistics, in arrests per 100,000 residents for assault, murder, and rape in each of the 50 US states in 1973. Also given is the percent of the population living in urban areas.*

Format *A data frame with 50 observations on 4 variables.*

- **Murder**: numeric Murder arrests (per 100,000)
- **Assault**: numeric Assault arrests (per 100,000)
- **UrbanPop**: numeric Percent urban population
- **Rape**: numeric Rape arrests (per 100,000)

To access a specific variable inside of a dataset, you will use the *$* operator between the dataset name and the variable name:

```
summary( USArrests$Murder )
summary( USArrests$Assault )
```

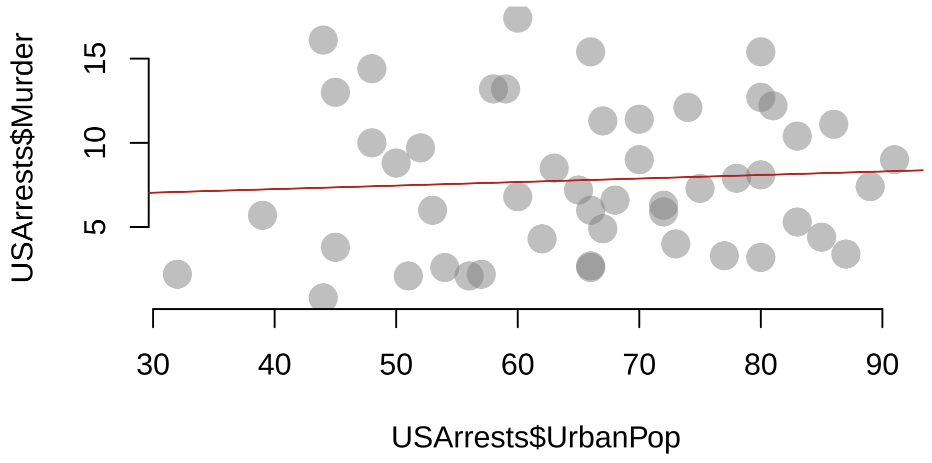
Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0.8	4.075	7.25	7.788	11.25	17.4

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
45	109	159	170.8	249	337

Is there a relationship between urban density and crime?

```
plot( USArrests$UrbanPop, USArrests$Murder,
      frame.plot=F, pch=19, cex=2,
      col=gray( level=0.5, alpha=0.5 ) )

abline( lm( USArrests$Murder ~ USArrests$UrbanPop ), col="firebrick" )
```



2.5.2 Using the Lahman Data

Let's take a look at some of the data available in the Lahman package.

```
data( package = "Lahman" ) # All datasets in package "Lahman":
```

DATASET NAME	DESCRIPTION
AllstarFull	AllstarFull table
Appearances	Appearances table
AwardsManagers	AwardsManagers table
AwardsPlayers	AwardsPlayers table
AwardsShareManagers	AwardsShareManagers table
AwardsSharePlayers	AwardsSharePlayers table
Batting	Batting table
BattingPost	BattingPost table
CollegePlaying	CollegePlaying table
Fielding	Fielding table
FieldingOF	FieldingOF table
FieldingOFsplit	FieldingOFsplit table
FieldingPost	FieldingPost data
HallOfFame	Hall of Fame Voting Data
HomeGames	HomeGames table
LahmanData	Lahman Datasets
Managers	Managers table
ManagersHalf	ManagersHalf table
Parks	Parks table
People	People table
Pitching	Pitching table
PitchingPost	PitchingPost table
Salaries	Salaries table
Schools	Schools table
SeriesPost	SeriesPost table
Teams	Teams table
TeamsFranchises	TeamFranchises table
TeamsHalf	TeamsHalf table
battingLabels	Variable Labels
fieldingLabels	Variable Labels
pitchingLabels	Variable Labels

We see that we have lots of datasets to choose from here. I will use the **People** dataset, which is a list of all of the Major League Baseball players over the past century and their personal information.

```
library( Lahman )     # loads Lahman package
data( People )        # loads the People dataset from Lahman
head( People )        # preview dataset
```

Here are some common functions for exploring datasets:

```
names(   People )      # variable names
nrow(    People )      # number of players (rows) in dataset
summary( People )      # descriptive statistics for each variable
```

We can use **help(People)** to get information about the dataset, including a data dictionary.

```
help( People )  # players dataset
```

Start helpfile:

R: People table

People table

Description

People table - Player names, DOB, and biographical info. This file is to be used to get details about players listed in the Batting, Pitching, and other files where players are identified only by playerID.

Usage

Format

A data frame with 20370 observations on the following 26 variables.

playerID

A unique code assigned to each player. The playerID links the data in this file with records on players in the other files.

birthYear

Year player was born

birthMonth

Month player was born

birthDay

Day player was born

birthCountry

Country where player was born

birthState

State where player was born

birthCity

City where player was born

deathYear

Year player died

deathMonth

Month player died

deathDay

Day player died

deathCountry

Country where player died

deathState

State where player died

deathCity

City where player died

nameFirst

Player’s first name

nameLast

Player’s last name

nameGiven

Player’s given name (typically first and middle)

weight

Player’s weight in pounds

height

Player’s height in inches

bats

a factor: Player’s batting hand (left (L), right (R), or both (B))

throws

a factor: Player’s throwing hand (left(L) or right(R))

debut

Date that player made first major league appearance

finalGame

Date that player made first major league appearance (blank if still active)

retroID

ID used by retrosheet, https://www.retrosheet.org/

bbrefID

ID used by Baseball Reference website, https://www.baseball-reference.com/

birthDate

Player's birthdate, in as.Date format

deathDate

Player's deathdate, in as.Date format

Details

debut, finalGame were converted from character strings with as.Date.

Source

Lahman, S. (2023) Lahman's Baseball Database, 1871-2022, 2022 version, https://www.seanlahman.com/baseball-archive/statistics/

Examples

End helpfile

Chapter 3

The R Language

3.1 Key Concepts

R is a specialized programming language created by statisticians for data analysis and visualization.

- R Console
- Base R
- Packages
- Comprehensive R Archive Network (CRAN)

3.2 R: An Open Source Language for Statistical Computing

R is a language that was designed for **statistical computing**, the art of combining computer science tools for problem-solving with models from statistics. The goal is to turn raw data into useful, actionable insights. This field has come to be known as **Data Science**.

R is an **open source** language, which means that applications built in R are not only free, but users are allowed to access and modify the *source code*.

As a result of this design approach, it is extremely easy to develop and adapt code in R. Because of the freedom this provides, R users have expanded the power and functionality of **Core R** for nearly a quarter century.

Custom applications and tools that users create for R are called **packages** (also called **libraries** when you are loading them). Packages are programs designed to perform a specific type of analysis or visualization.

The best part of R is how easy it is to access cutting edge software by installing new packages in a two lines of code:

```r
install.packages( "tidyverse" )   # install the package
library( "tidyverse" )            # load the package
```

Popular R Packages: [**A RECENT LIST**]

3.3 R as a Social Network

The R Foundation is a nonprofit that maintains the R language and ensures it remains free and accessible to everyone in the world. Packages are shared through the Comprehensive R Archival Network (The **CRAN**), a group of servers housed primarily at universities that store R packages so they can be quickly downloaded and deployed.

There are over 15,000 packages that users have created for R. They perform a wide variety of tasks such as data preparation, specialized statistical analysis, custom data visualizations, or specific analytical tasks such as text analysis or network analysis.

This functionality is a primary reason R has become one of the most popular languages used by academics and data scientists. Provides a very simple way for people to develop cool tools and share them with the world. It became popular because it was built by smart and creative people, who attracted other smart and creative people, who created cool tools, which then attracted more smart and creative people. **R Package Downloads**

3.4 R as an Operating System

R is a **programming language**. We can think of a programming languages as instructions that are evaluated and carried out by a computer. R, then, is simply one way to give instructions to computers.

This is a limited view of R, though. It is better understood as an operating system for data science software. Just as Windows allows you to turn on your computer, open a web browser, moved files around, and write a paper using MS Word, R allows you to access the CRAN, install and run packages, and manage files while organizing large data projects. Just like Windows would be a very boring piece of software without all of the applications you run while on the computer, R would be a boring language without all of the packages it can run.

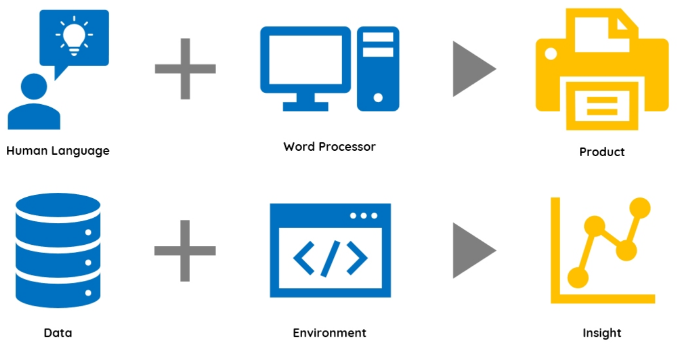


Figure 3.1: *R is both a programming language and programming environment.*

3.5 Downloading & Installing R

You can download and install R quickly and easily from the **Comprehensive R Archive Network**, or **CRAN**. It is a decentralized website that's hosted and updated by academic institutions all over the world. In other words, R would survive a semi-global catastrophic event. It contains:

- The latest version and past versions of R
- Extensions, also called **packages**, for R
- Package and version documentation
- Books, blogs, conferences, news, etc.

3.6 R Console

After installing, when you open Base R directly you will see the **command-line interface**, or a **console**. This is used to type R code is directly evaluated by the environment, a process known as working *interactively*.

While this is practice is a quick way to run some simple code, it is difficult to develop complex programs in real-time (it would be like writing a play while it is being acted out). A more typical and organized way to create data recipes is through **scripting**, which we address below.

3.7 Extending R's Functionality: Packages

Packages are collections of new commands, a.k.a. **functions**, that are developed and shared by the worldwide R userbase. Packages greatly expand the power and functionality of **base R**, the "vanilla" or unmodified version of R.

While CRAN is the most popular package archive, others include Bioconductor and GitHub.

If R were the Constitution of a nation, packages would be its amendments - they not only provide more freedom for the user, they address new ideas and practices that were unforeseen by R's founders. More on packages:

- Functions and packages are developed in response to identified needs
- If your needs are unmet by base R, there's likely a package for it
- Altogether, there are over 18,000 packages on CRAN, alone
- There are tens of thousands of unpublished packages
- Entire ecosystems of packages exist, e.g, *Tidyverse*

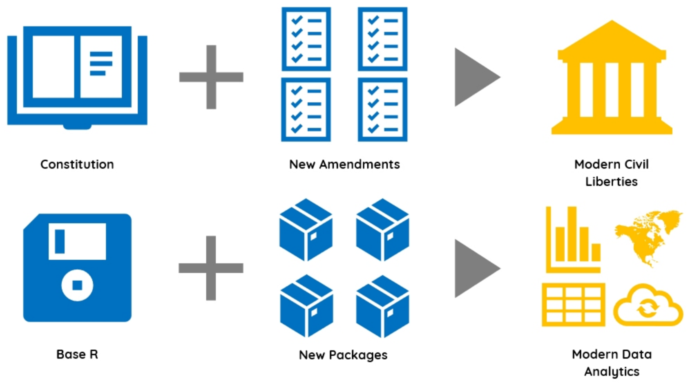


Figure 3.2: *Packages give users more freedom and resolve issues unforeseen by R's founders.*

You can **install packages** in R by calling the `install.packages()` function, *with the package name in quotations*:

```
install.packages("my_package")
```

Once installed, you can **load packages** by calling the `library()` function, *without quotations.*

Note: You only need to install a package *once*. However, you must load each package *every time you start R*:

```
library(my_package)
```

Note: "Packages" and "libraries" are two words for the same thing. They both refer to a set of **functions** that have been "packaged" or are organized into a "library" to be shared.

Fun Fact: R is an implementation of an older programming language, **S**. John Chambers first developed S in 1976 to make statistical analysis a point-and-click, interactive, and user-friendly process. However, Chambers' underlying philosophy reflects the use of R packages to this day:

> "We wanted users to be able to be in in an interactive environment, where they did not consciously think of themselves as programming. Then as their needs became clearer and their sophistication increases, they should be able to slide gradually into programming, when the language and system aspects would become more important."

3.8 Resources

There's a litany of online and print resources introducing the R language. Here are a few that we find instructive:

I) Full-Length Introductions to R:

- "Part I: Foundations, Introduction to R" (Lecy, 2018)
- "Intro to R: Nuts * Bolts" (Crawford, 2018)

II) Publications & Articles:

- "What is R? Introduction to R and the R Environment" (CRAN, 2001)
- "R: A Language for Data Analysis and Graphics" (Ihaka & Gentleman, 1996)

III) Handouts & Cheat Sheets:

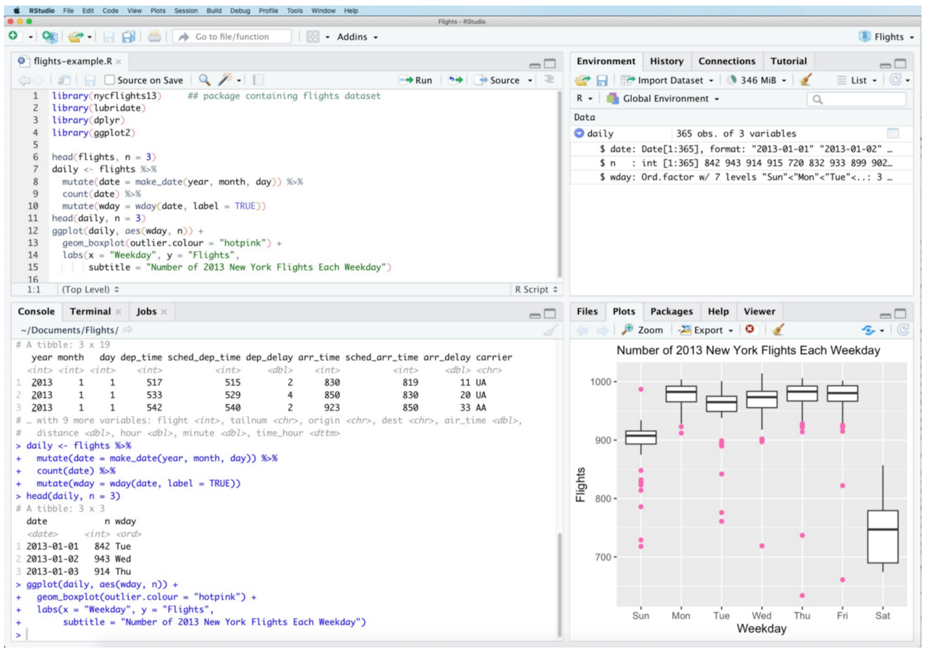
- "R: Some Helpful Vocabulary" (Lecy, 2017)
- "Base R Cheat Sheet" (RStudio, 2016)

IV) Videos:

- "R in 60 Seconds" (Lecy, 2018)
- "John Chambers Interview [On the History of S & R]" (Statistical Learning, 2013)

Chapter 4

R Studio



```r
library(nycflights13)     ## package containing flights dataset
library(lubridate)
library(dplyr)
library(ggplot2)

head(flights, n = 3)
daily <- flights %>%
  mutate(date = make_date(year, month, day)) %>%
  count(date) %>%
  mutate(wday = wday(date, label = TRUE))
head(daily, n = 3)
ggplot(daily, aes(wday, n)) +
  geom_boxplot(outlier.colour = "hotpink") +
  labs(x = "Weekday", y = "Flights",
       subtitle = "Number of 2013 New York Flights Each Weekday")

1:1   (Top Level)                                            R Script
```

```
Data
daily              365 obs. of 3 variables
    $ date: Date[1:365], format: "2013-01-01" "2013-01-02" …
    $ n   : int [1:365] 842 943 914 915 720 832 933 899 902…
    $ wday: Ord.factor w/ 7 levels "Sun"<"Mon"<"Tue"<..: 3 …
```

```
# A tibble: 3 x 19
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time arr_delay carrier
  <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>     <dbl> <chr>
1  2013     1     1      517            515         2      830            819        11 UA
2  2013     1     1      533            529         4      850            830        20 UA
3  2013     1     1      542            540         2      923            850        33 AA
# … with 9 more variables: flight <int>, tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
#   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
> daily <- flights %>%
+   mutate(date = make_date(year, month, day)) %>%
+   count(date) %>%
+   mutate(wday = wday(date, label = TRUE))
> head(daily, n = 3)
# A tibble: 3 x 3
  date           n wday
  <date>     <int> <ord>
1 2013-01-01   842 Tue
2 2013-01-02   943 Wed
3 2013-01-03   914 Thu
> ggplot(daily, aes(wday, n)) +
+   geom_boxplot(outlier.colour = "hotpink") +
+   labs(x = "Weekday", y = "Flights",
+        subtitle = "Number of 2013 New York Flights Each Weekday")
> 
```

4.1 Key Concepts

This chapter introduces RStudio, a Graphical User Interface (GUI) that makes it easier to use powerful features in R and manage large projects.

New vocabulary:

- Integrated Development Environment (IDE)
- RStudio Panes:

    1. Console
    2. Source
    3. Plots
    4. Viewer
    5. Environment
    6. History
- Workspace & Global Environment

4.2 What is RStudio?

Recall that R is both a language and an environment. **RStudio** is an **Integrated Development Environment**, or **IDE**, which is an enhanced, feature-rich programming environment with an easy-to-use **graphical user interface**, or **GUI**. While the base R environment is mostly text, RStudio has intuitive icons (hence, "graphical") for point-and-click, automated operations.

RStudio's layout is comprised of a menu, console, and a series of **panes**, or windows in the RStudio interface. Most panes are feature-rich and all panes serve a key purpose, but we'll only focus on the most critical panes for getting started in RStudio.

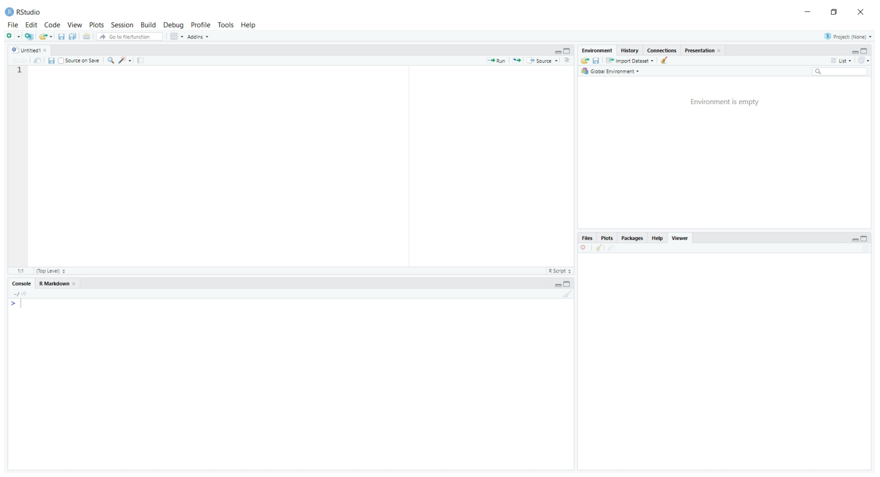


Figure 4.1: *The RStudio environment with four open panes: The Source, Console, Environment, and Viewer panes.*

4.3 Downloading & Installing RStudio

RStudio requires R 3.0.1+. If you don't already have R, download it here.

DOWNLOAD R

RStudio is free, open source, and easy to install. Select the *Desktop* edition on their download page:

DOWNLOAD R STUDIO

4.4 A Tour of RStudio

RStudio is comprised of a main menu and a series of panes, each with their own purpose and features. We focus on the following:

1. Console Pane
2. Source Pane
3. Plots Pane
4. Viewer Pane
5. Environment Pane
6. History Pane

4.5 The Console Pane

The **console pane** is where R expressions are evaluated. In other words, this is where your code is executed. Recall that working in-console is also known as working *interactively* and, typically, working in-console is more often for "quick and dirty" tasks, like printing contents of your working directory.

The Console Pane

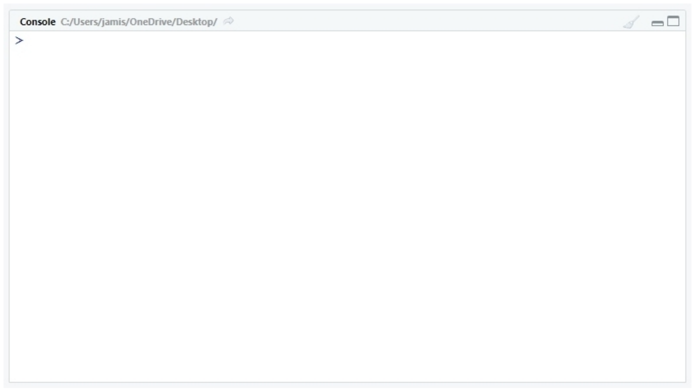


Figure 4.2: *The console is where code is executed and is typically used for "quick and dirty" tasks.*

The console panel lists your current working directory. Notably, even when using point-and-click mechanics to, e.g. import data or change directories, the code for such tasks will still execute in the console. Such click-to-code operations are called **macros**.

Pro Tip: When writing a script, especially when writing out new directory paths, it's sometimes quicker to use use a click-to-code operation and simply copy and paste the macro code from the console to your script.

4.6 The Source Pane

The **source pane** contains any opened scripts. In starting a new R session, this pane isn't visible until you've opened a new or preexisting script. Multiple scripts may be opened at one time and are navigable using tabs along the top of the source pane.

The Source Pane

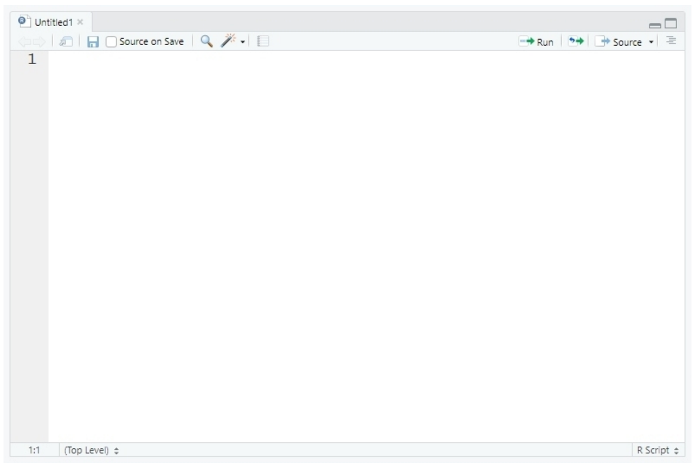


Figure 4.3: *The source pane contains opened scripts. It won't appear until you've opened at least one.*

Depending on the type of script, e.g. plain text scripts (`.R`), publications (`.Rmd`), presentations (`.Rpres`), and apps, each script provides different options in the pane's toolbar. Common options include:

- **Show in New Window:** Open the script in a separate window; valuable for two or more monitors
- **Save Current Document:** Update an existing script or title and save a new script
- **Find/Replace:** Both conventional and advanced options to find and replace code
- **Run:** Run the line of code where the cursor is, or multiple lines if highlighted
- **Show Document Outline:** View (and jump to) script's table of contents

4.7 Plots, Viewer, & Help Panes

The **Plots**, **Viewer**, and **Help panes** are used to viewing data visualizations, HTML output, and helpful documentation.

4.7.1 The Plots Pane

The **Plots pane** allows users to view, export, and publish non-interactive data visualizations. R uses the built-in `graphics` package by default, but a variety of packages exist such as `lattice` and `ggplot2`. While the output displayed is not interactive, it is *responsive*, i.e it will re-render its scales appropriately if you change the height or width of a plot. Notably, the "Zoom" option opens visualizations in a new window, while the "Export" option allows you to save the image with user-defined dimensions and in a variety of formats.

4.7.2 The Viewer Pane

The **Viewer pane** renders interactive graphics in HTML with the same options as the **Plots pane**. Brevity aside, it's awesome.

The Plots & Viewer Panes

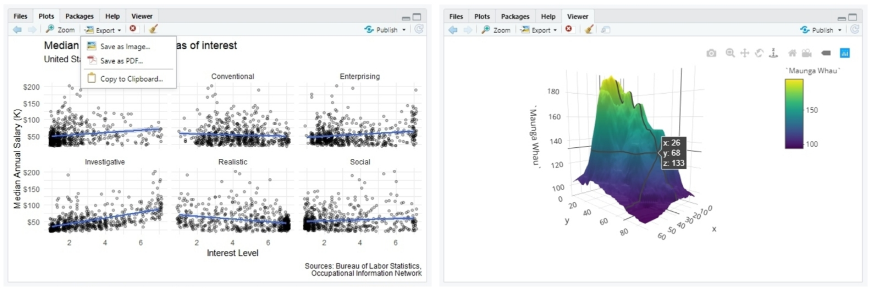


Figure 4.4: *The Plots and Viewer panes visualize non-interactive and interactive graphics, respectively.*

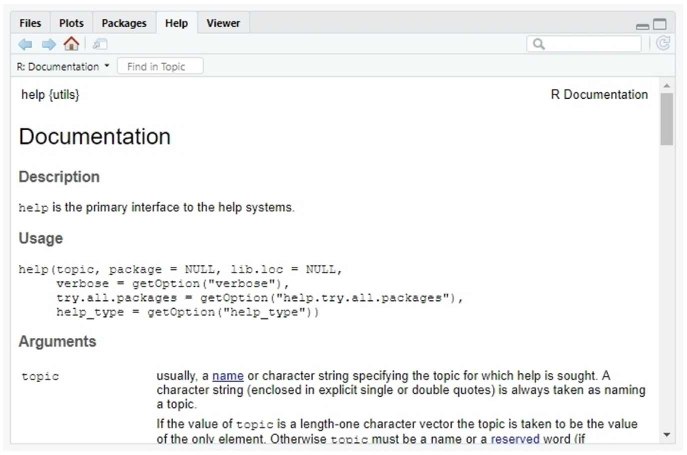
4.7.3 The Help Pane

The **Help pane** is one of the most valuable panes for any R user. By calling function `help()` with a dataset, package, or bare function name (i.e. a function name without `()`), its documentation, if available, appears here.

Note: Unless you're using external data or custom functions, there's almost always documentation. Whether it's the unit of measurement for a variable in a dataset or the limits you can specify for a function argument, this little nook in RStudio is invaluable to new and advanced users, alike.

Pro Tip: Instead of the `help()` function, you can use the `?` function before an object name, e.g. `?install.packages`.

The Help Pane



help {utils} R Documentation

Documentation

Description

`help` is the primary interface to the help systems.

Usage

```
help(topic, package = NULL, lib.loc = NULL,
     verbose = getOption("verbose"),
     try.all.packages = getOption("help.try.all.packages"),
     help_type = getOption("help_type"))
```

Arguments

`topic` usually, a name or character string specifying the topic for which help is sought. A character string (enclosed in explicit single or double quotes) is always taken as naming a topic.

If the value of `topic` is a length-one character vector the topic is taken to be the value of the only element. Otherwise `topic` must be a name or a reserved word (if

Figure 4.5: *The invaluable Help pane displays documentation for packages, functions, and datasets.*

4.8 The Environment and History Panes

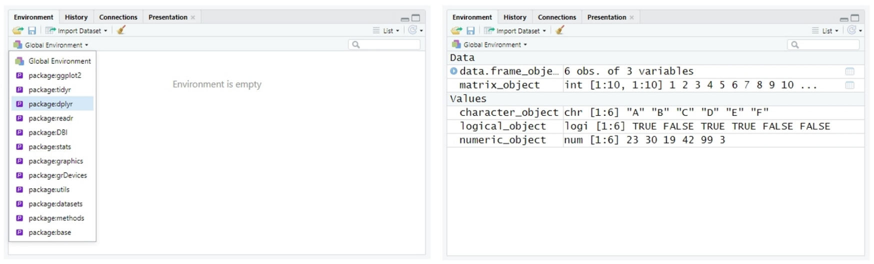
The **Environment** and **History panes** display the objects in your environment and the history of your in-console commands.

4.8.1 The Environment Pane

Again, R is both a language and an environment. The **Environment pane** displays objects that are stored within your session's **workspace**, or **global environment**, which must be recreated or reloaded with each new session. Note the following options:

- **Environment:** Opens a dropdown menu to select different environments, e.g. package environments
- **Load Workspace:** Opens a file explorer to load previously saved workspaces and their objects
- **Import Datasets:** Opens a dropdown menu to read in datasets that you can store in objects
- **Clear Objects from Workspace:** Removes all objects stored in the global environment

The Environment Pane



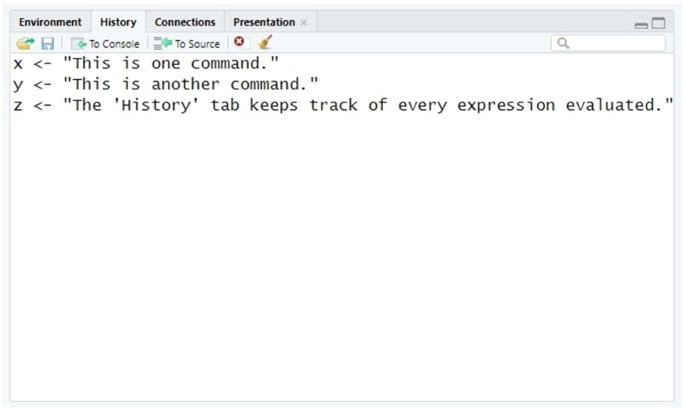
```
Data
data.frame_obje… 6 obs. of 3 variables
matrix_object    int [1:10, 1:10] 1 2 3 4 5 6 7 8 9 10 ...
Values
character_object chr [1:6] "A" "B" "C" "D" "E" "F"
logical_object   logi [1:6] TRUE FALSE TRUE TRUE FALSE FALSE
numeric_object   num [1:6] 23 30 19 42 99 3
```

Figure 4.6: *The environment pane displays any objects you've imported, loaded, or stored in your global environment.*

4.8.2 The History Pane

The **History pane** documents every command you've executed in your session. When you select a line, you can paste it directly into the console pane with "To Console" or directly into the source pane with "To Source".

The History Pane



```
x <- "This is one command."
y <- "This is another command."
z <- "The 'History' tab keeps track of every expression evaluated."
```

Figure 4.7: *The history pane records every command you've run in your session.*

4.9 Customizing Your Pane Layout

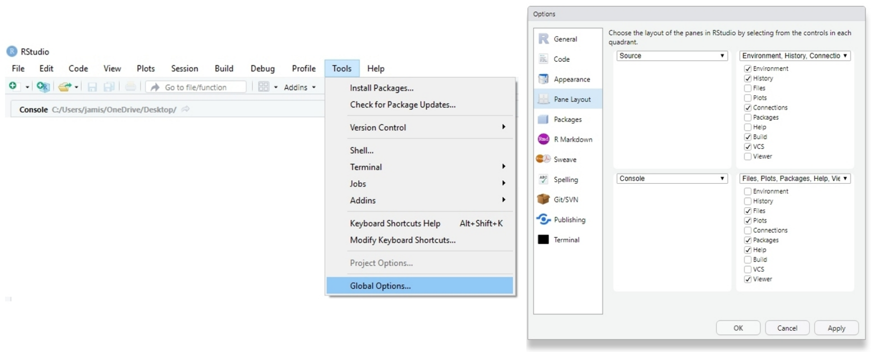
In RStudio, you can customize both where panes are displayed as well as which panes to show by default.

4.9.1 Layouts for Beginners: Taking Great Panes

Panes cannot be removed entirely from the RStudio interface, you but can shuffle them by order of importance. Click the "Tools" dropdown in themenu, "Global

Options…”, and “Pane Layout”. We recommend focusing on those discussed in this chapter.

Customizing Pane Layout



Choose the layout of the panes in RStudio by selecting from the controls in each quadrant.

Source

Environment, History, Connectio
- ✓ Environment
- ✓ History
- ☐ Files
- ☐ Plots
- ✓ Connections
- ☐ Packages
- ☐ Help
- ✓ Build
- ✓ VCS
- ☐ Viewer

Console

Files, Plots, Packages, Help, Vie
- ☐ Environment
- ☐ History
- ✓ Files
- ✓ Plots
- ☐ Connections
- ✓ Packages
- ✓ Help
- ☐ Build
- ☐ VCS
- ✓ Viewer

Figure 4.8: *You can customize which panes appear, and where, in Global Options’ Pane Layout.*

4.9.2 Less Important Panes, or Panes in the Rear

There are a couple of panes worth mentioning for new users. However, they are seldom used by advanced users:

- **Files:** Set working directories and create, copy, rename, and delete folders
- **Packages:** Install, load, update, unload, and uninstall added (“User Library”) and built-in (“System Library”) packages

Pro Tip: You probably won’t use these panes often. One of the benefits of scripted languages is that they can be reproduced by other users. As a rule, since much of your work will require loading packages, you should include the `library()` function with script-dependent packages at the start of every work. The same applies to working directories with the `setwd()` function.

4.9.3 Expanding to Fullscreen: Focus on the Pane

If you want to expand a pane, or “zoom”, to fullscreen mode, select “View” in the RStudio menu and “Panes”.

Toggling Fullscreen Panes

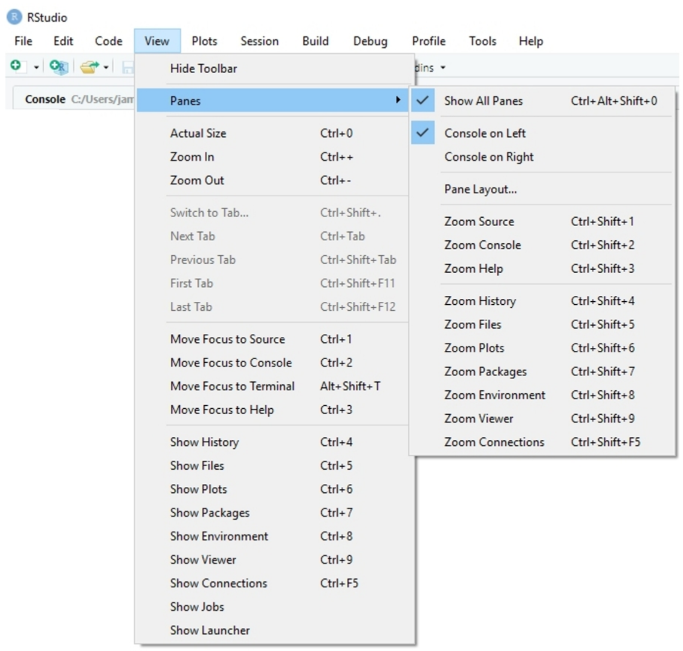


Figure 4.9: *You can expand any pane to fullscreen mode in the "View" dropdown of the RStudio menu.*

4.10 The RStudio Menu: File, Session, & Help

The RStudio menu allows you to do virtually everythign we've seen in each pane and more. The following tours a few key menu sections we've not yet seen, including how to open new scripts, handle sessions, and access R-related cheat sheets.

4.10.1 The File Submenu: Saving & Loading

The **File** submenu is the start of every scripted data product in RStudio. Just select "New File" and a litany of possible data products, about which we'll learn more over time, are available to open. Remaining options relate to the saving and loading of scripts, projects, and datasets.

The File Submenu

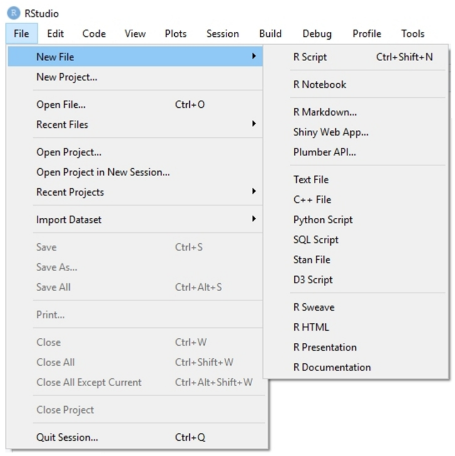


Figure 4.10: *"File" handles all new, saving, and loading operations.*

4.10.2 The Session Submenu: Sessions, Directories, & Restarting R

The **Session** submenu is a critical part of any R session. While you can always recreate a session by recreating objects, you can save computing time by loading saved session file. This submenu also allows you to set your working directory.

Note: There are times when you just have to restart R - maybe you started an infinite recursion loop or maybe you attempted to read in the data from a month's worth of Harrier Jet landings to local memory - if so, the "Restart R" option is here for you.

The Session Submenu

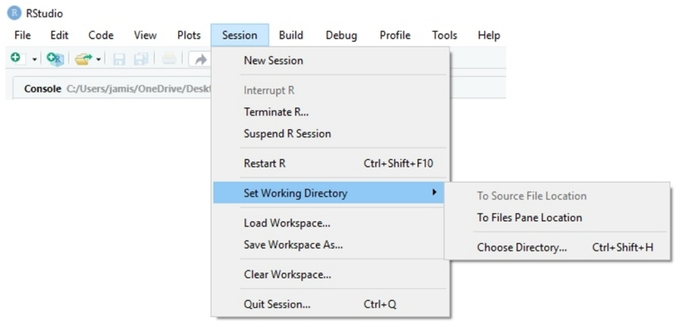


Figure 4.11: *Save and load sessions, restart R, or handle directories in the session submenu.*

4.10.3 The Help Submenu: Cheat Sheets

The **help menu** has one main draw (for now): **cheat sheets**. Selecting a cheat sheet will automatically download a cheat sheet on an R-related topic of your choosing. Typically, cheat sheets summarize RStudio related packages and data products, but there is one for base R. For example, you can find the RStudio IDE cheat sheet here.

The Help Submenu

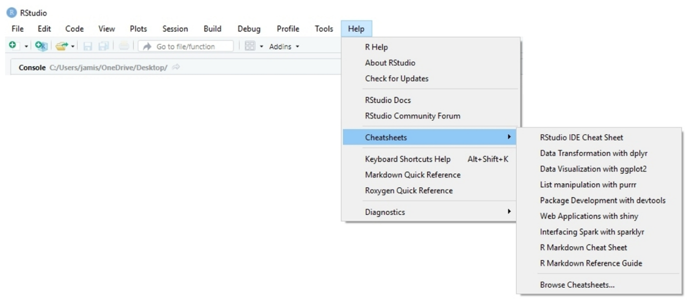

RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Console C:/Users/jamis/OneDrive/Desktop/
R Help
About RStudio
Check for Updates
RStudio Docs
RStudio Community Forum
Cheatsheets
Keyboard Shortcuts Help Alt+Shift+K
Markdown Quick Reference
Roxygen Quick Reference
Diagnostics
RStudio IDE Cheat Sheet
Data Transformation with dplyr
Data Visualization with ggplot2
List manipulation with purrr
Package Development with devtools
Web Applications with shiny
Interfacing Spark with sparklyr
R Markdown Cheat Sheet
R Markdown Reference Guide
Browse Cheatsheets...

Figure 4.12: *Grab cheat sheets on the fly from the Help submenu.*

4.11 Global Options: Aesthetic & Functional Preferences

Global options are accessed in the "Tools" submenu and allow users to modify their RStudio interface in myriad ways, both aesthetically and functionally. We recommend new users experiment with these options and visit a few notable modifications.

Accessing Global Options

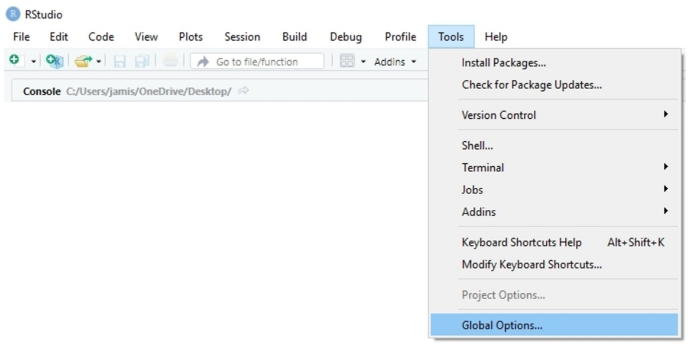

RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Console C:/Users/jamis/OneDrive/Desktop/
Install Packages...
Check for Package Updates...
Version Control
Shell...
Terminal
Jobs
Addins
Keyboard Shortcuts Help Alt+Shift+K
Modify Keyboard Shortcuts...
Project Options...
Global Options...

Figure 4.13: *Access "Global Options" in the "Tools" submenu.*

4.11.1 Code: Autoformatting & Behavior

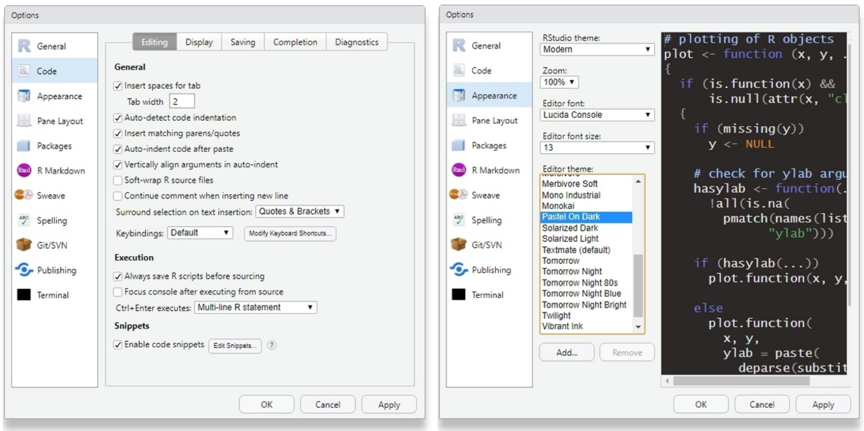
The "Code" section affects the way R automatically formats your code and how you choose to write and run it. It has sensible defaults, many of which you may not be prepared to appreciate quite yet. For now, consider the following:

- **Indentating:** In the "Editing" tab, consider a "tab width" that works best for you. When indenting, would you prefer two characters (i.e. spaces), or four? The former allows more compact code. The latter allows for more intepretable code.
- **Guide Margin:** In the "Display" tab, consider applying a "margin column" of 80 or 100 characters (i.e. spaces). This creates a subtle guide in your scripts that helps keep code concise and readable. Even basic code within basic code can create, what RStudio's Chief Scientist Hadley Wickham refers to as, "Dagwood sandwich" code.

4.11.2 Appearance: Express Yourself Intepretably

The **Appearance** section allow you to customize the size, font, and color of your text as well as the "theme" colors of your RStudio interface. Here, "theme" is both functional and aesthetic. For example, darker themes are more conducive to night owls. For all themes, certain syntax uses different colors to make your code more interpretable - keep this in mind for each theme!

Notable Global Options: Code & Appearance



General

[✓] Insert spaces for tab
Tab width: 2
[✓] Auto-detect code indentation
[✓] Insert matching parens/quotes
[✓] Auto-indent code after paste
[✓] Vertically align arguments in auto-indent
[ ] Soft-wrap R source files
[ ] Continue comment when inserting new line
Surround selection on text insertion: Quotes & Brackets
Keybindings: Default

Execution

[✓] Always save R scripts before sourcing
[ ] Focus console after executing from source
Ctrl+Enter executes: Multi-line R statement

Snippets

[✓] Enable code snippets

RStudio theme: Modern
Zoom: 100%
Editor font: Lucida Console
Editor font size: 13
Editor theme: Merbivore Soft, Mono Industrial, Monokai, [✓ Pastel On Dark], Solarized Dark, Solarized Light, Textmate (default), Tomorrow, Tomorrow Night, Tomorrow Night 80s, Tomorrow Night Blue, Tomorrow Night Bright, Twilight, Vibrant Ink

```
# plotting of R objects
plot <- function (x, y, .
{
  if (is.function(x) &&
      is.null(attr(x, "cl
  {
    if (missing(y))
      y <- NULL

    # check for ylab argu
    hasylab <- function(.
      !all(is.na(
        pmatch(names(list
              "ylab")))

    if (hasylab(...))
      plot.function(x, y,

    else
      plot.function(
        x, y,
        ylab = paste(
          deparse(substit
```

Figure 4.14: *Global options allow you to set preferences that can significantly impact your experience with RStudio.*

4.12 Further Resources

The following resources are helpful in learning more about RStudio and coding conventions:

I) Full Introductions to RStudio

- "What are R and RStudio?" (Ismay & Kim, 2019)
- "Intro to R: Nuts & Bolts" (Crawford, 2018)

II) About RStudio

- RStudio Homepage (RStudio, 2019)
- RStudio About Page (RStudio, 2019)
- RStudio Product Page (RStudio, 2019)

III) Conventions

- "The State of Naming Conventions in R" (Baath, 2012)

Chapter 5

R: An Overdesigned Calculator

5.1 Introduction

Chief among R's many capabilities are basic mathematical operations, making it a severely overdesigned calculator. The only thing a TI-84 has on R is that you can't play Super Mario Brothers with the latter - *yet* (package developers… if your listening…).

So what?

Arithmetic operations in R are key to transforming your data, whether calculating property code violations *per capita* or converting U.S. dollars to Indian rupees. In short, you use them to make new variables from existing ones. When we combine arithmetic with objects that store one or more values, we're dangerously close to practicing algebra.

5.2 Key Concepts

In this chapter, we'll learn and practice basic arithmetic functions in R, assign calculations and their resulting values to objects, and use those objects in algebraic operations. Key concepts include:

- Arithmetic Operators
- Operator Precedence
- Assignment

5.3 Key Takeaways

Everything you need to know in a few bullet points:

- Arithmetic operators include: `+`, `-`, `*`, `/`, `^`, `( )`
- Calculations follow the **order of operations**
- Create **objects** with **assignment**:
  - `x` stores the value `3` after calling `x <- 3`
- Numeric bjects act like variables in algebra:
  - `x + 2` equals `5`

5.4 Arithmetic Operators

"What sort of free will is left when we come to tabulation and arithmetic? When it will all be a case of twice two makes four?

You don't need free will to determine that twice two is four." (Dostoyevsky)

You remember arithmetic, right? That peculiar field of mathematics in which people who admittedly "don't math", for whatever reason, actually use every day?

Arithmetic operators in R work just like they did in primary school, including addition, subtraction, multiplication, division, and exponentiation:

- `+` or **addition**, e.g. `2 + 2`
- `-` or **subtraction**, e.g. `2 - 2`
- `*` or **multiplication**, e.g. `2 * 2`
- `/` or **division**, e.g. `2 / 2`
- `^` or **exponentiation**, e.g. `2 ^ 2`
- `( )` for **order of operations**, e.g. `((2 + 2) * 2)`

The following example has a number of operations. Run the code to see what happens:

```
7 + 3       # Addition

8 - 12      # Subtraction

9 * 9       # Multiplication

10 / 3      # Division

10 ^ 3      # Exponentiation
```

Eureka! Forget your mobile phone's calculator app. Install R on it!

(Plus, you can browse Reddit during class, but it looks like you're working).

5.4.1 Your Turn

Instructions: Perform the following arithmetic operation in R.

Tip: Numeric values in R *don't use commas.*

```
# Raise 5 to the fifth power

# Subtract 30 from 100

# Divide 1,000 by 300
```

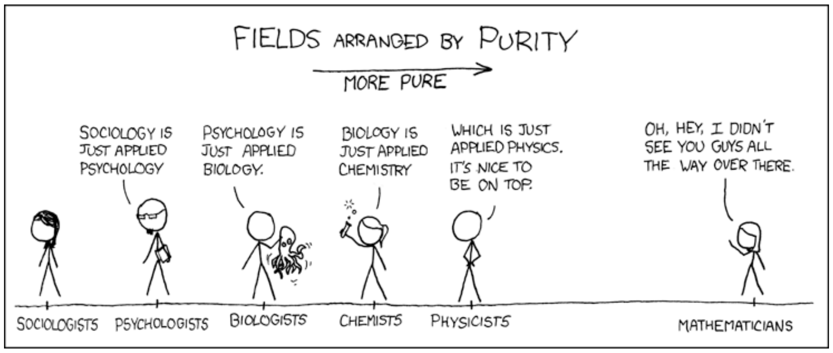

FIELDS ARRANGED BY PURITY
MORE PURE
SOCIOLOGY IS JUST APPLIED PSYCHOLOGY
PSYCHOLOGY IS JUST APPLIED BIOLOGY.
BIOLOGY IS JUST APPLIED CHEMISTRY
WHICH IS JUST APPLIED PHYSICS. IT'S NICE TO BE ON TOP.
OH, HEY, I DIDN'T SEE YOU GUYS ALL THE WAY OVER THERE.
SOCIOLOGISTS
PSYCHOLOGISTS
BIOLOGISTS
CHEMISTS
PHYSICISTS
MATHEMATICIANS

Figure 5.1: *Arithmetic operators are the most atomic functions in R programming.*

Source: XKCD

5.5 Order of Operations

Here's another blast from the past: the **operator precedence**. At least, that's what it's called in programming languages. You probably remember it as the **order of operations**.

If you don't recall the specific rules, perhaps you remember the mnemonic devices: **PEMDAS** or **Please Excuse My Dear Aunt Sally**. We are not sure know what Aunt Sally did, but we are pretty sure she deserves whatever punshinment she received.

PEMDAS reminds us *the order arithmetic operations are evaluated*, a.k.a. the **order of operations**:

1. **Parenthesis**, or expressions inside `( )`
2. **Exponents**, or raising one value to the power of another with `^`
3. **Multiplication**, or multiplying values with `*`
4. **Division**, or dividing values with `/`
5. **Addition**, or adding values with `+`

6. **Subtraction**, or subtracting values with `-`

Arithmetic operations in R are also evaluated in the same order. Can you guess the results before evaluating the expressions? Press “Run” to execute the code and see the results:

```
# easy one
5 + 10 / 5

# harder
5 - 10 + 5
5 + 10 - 5

# similarly
5 * 10 / 5
5 / 10 * 5

# easier
3 * 2 ^ 2

# hmmm
2 * (2 + 3) * 3

# huh?
( 2 + 3 ) / 5 * 2
( 2 + 3 ) * 5 / 2
( 2 + 3 ) / 2 * 5
```

Note that R is indifferent to order of operations for addition vs subtraction, and multiplication vs division.

For cases where both occur the code is just executed from left to right.

5.5.1 Your Turn

Instructions:

The formula to calculate a monthly mortgage payment based upon the loan amount, annual interest rate, and loan term (in months) is calculated as follows:

$$PAYMENT = \frac{principal \cdot \frac{interest\_rate}{12}}{1 - (1 + \frac{interest\_rate}{12})^{-\ months}}$$

Let’s say we have a $100,000 loan at a 5% interest rate amortized over 360 months (30 years). The payments would be as follows:

$$PAYMENT = \frac{100k \cdot \frac{0.05}{12}}{1 - (1 + \frac{0.05}{12})^{-\ 360}}$$

Can you type the formula into R correctly? The payment should come to $536.82 a month.

```
# Type your formula here
```

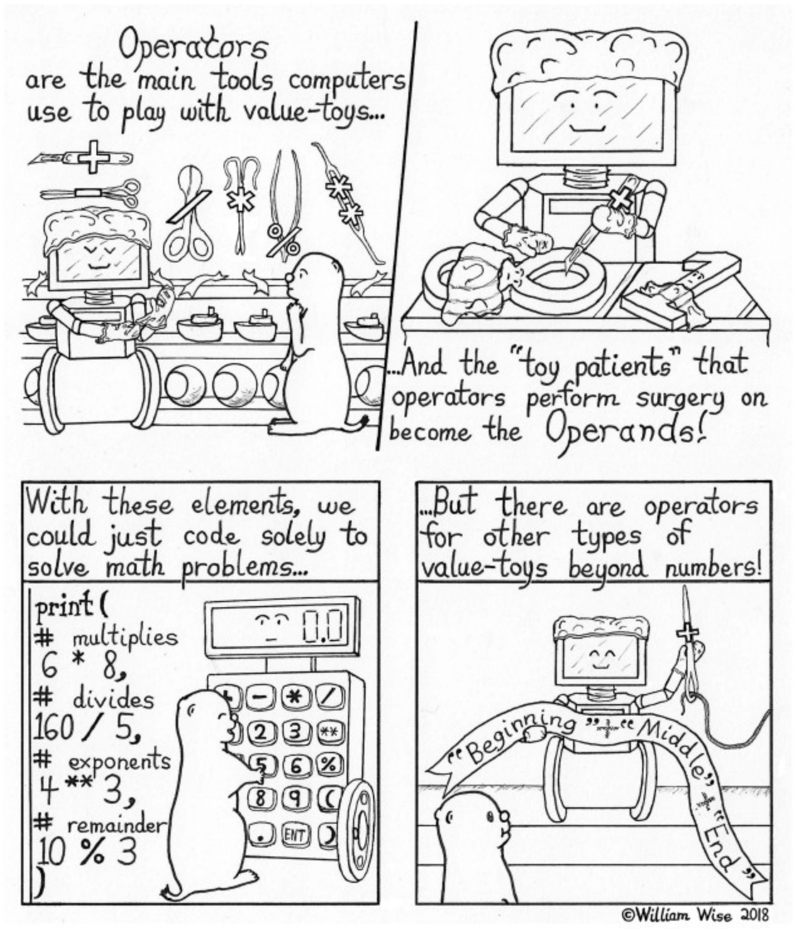

Operators are the main tools computers use to play with value-toys...
...And the "toy patients" that operators perform surgery on become the Operands!
With these elements, we could just code solely to solve math problems...
print(
# multiplies
6 * 8,
# divides
160 / 5,
# exponents
4 ** 3,
# remainder
10 % 3
)
0.0
...But there are operators for other types of value-toys beyond numbers!
"Beginning" + "Middle" + "End"
©William Wise 2018

Figure 5.2: *A different programming language, but you get the gist.*

Source: Prairie World Comics

Chapter 6

Assignment

> "The philosophical workers… have to fix and formalize some great existing body of valuations - that is to say…
>
> …creations of value, which have become prevalent, and are for a time called 'truths'…" (Nietzsche)

Objects are the meat and potatoes of R. They store information like datasets, function code, individual values, and even metadata, e.g. coefficents in linear models.

But how do we create an object?

We use **assignment** to create an object and store data in it. This requires the **assignment operator**, or the `<-` arrow.

Assignment can be used to store external data that was imported into R, to save a statistic that we calculated, or remember data we type into R ourselves.

```r
x <- read.csv( "FileName.csv" )

x <- mean( y )

x <- c("Adam","Susie","Tony")
```

Assign the values 3 and 4 to objects `x` and `y`, respectively. Note that typing an object's name will report the object values.

```r
x <- 3          # Assign 20 to "x"

y <- 4          # Assign 480 to "y"

x               # Will this print "x" or 3 ?
```

```
x * y
```

Like algebra, `x` and `y` are used to *represent* numeric values.

Unlike algebra, we can *store many values* in objects, including other objects!

```
x <- 20      # Assign 20 to "x"

y <- 40      # Assign 40 to "y"

z <- x + y   # Assign the sum of "x" and "y" to "z"

z            # Print z
```

What do you think the following will return?

```
x <- 20
y <- x
x <- 30

y   # print the value stored by y
```

Why does R use the `<-` for assignment?

Technically R allows you to use the equals `=` operator for **assignment** instead of using the arrow `<-`.

BUT YOU SHOULD NEVER DO IT!.

Assignment is important enough that the arrow helps keep your code easy to read.

Also, mind your spaces!

- x <- y is assignment
- x < - y is less than negative y

6.0.1 Your Turn

What will the following statements return?

```
x <- 5
x <- 10
x

x <- 5
x < - 10
x

5 <- x
```

```r
5

5 -> x
x

x = 5
x

5 = x
x

y <- 10
x <- y <- 3
x
```

6.1 Valid Object Names

When creating new objects there are both rules and conventions for naming them.

The rules are fairly simple:

1. R is case sensitive, so `b` and `B` are different objects.
2. Object names can include letters, periods and underscores.
3. Object names can include numbers, but cannot begin with a number.

```r
x.01 <- 99  # good
x_01 <- 99  # this works
01.x <- 99  # produces an error

.x <- 99    # this works
.1x <- 99   # this doesn't
_x <- 99    # oddly this doesn't
```

In general it is good to name objects so they are easy to remember. You can combine words using one of three conventions:

- Camel Caps
- Underscores
- Periods

```r
myData <- 99   # camel caps
my_data <- 99  # underscore
my.data <- 99  # periods
```

Some people have strong views on these. You should find something that works for you and be consistent.

HEY, MISS LENHART! I FORGOT EVERYTHING ABOUT ALGEBRA THE MOMENT I GRADUATED, AND IN 20 YEARS NO ONE HAS NEEDED ME TO SOLVE *ANYTHING* FOR X!
I *TOLD* YOU I'D NEVER USE IT!
IN YOUR *FACE!*
IT'S WEIRD HOW PROUD PEOPLE ARE OF NOT LEARNING MATH WHEN THE SAME ARGUMENTS APPLY TO LEARNING TO PLAY MUSIC, COOK, OR SPEAK A FOREIGN LANGUAGE.

Figure 6.1: *You can't do this anymore, sadly.*

Source: XKCD

6.2 Further Resources

The following resources are helpful in learning more about arithmetic operators in R:

- "Arithmetic Operators" (CRAN)
- "Intro to R: Operators" (Crawford, 2019)
- "Quick-R: Operators" (Kabacoff, 2017)

6.3 Works Cited

The Hangover (2009) The Hunger Games (2010)