Working with data

Robert B. Gramacy
Virginia Tech Department of Statistics

bobby.gramacy.com

Entering Data

For a small number of observations, entering data directly into R is an option. There are a couple of ways to do that.

As we've seen already, we can create new objects directly on the console.

```
> salary <- c(18700000, 14626700, 14137500, 13800000)
> position <- c("QB", "QB", "DE", "QB")
> team <- c("Colts", "Patriots", "Panthers")
> last <- c("Manning", "Brady", "Pepper", "Palmer")
> first <- c("Peyton", "Tom", "Julius", "Carson")
```

It is usually convenient to put these vectors together into a data frame.

```
> top5salaries <- data.frame(last, first,
+    team, position, salary)
> top5salaries
     last  first     team position   salary
1 Manning Peyton    Colts       QB 18700000
2   Brady    Tom Patriots       QB 14626700
3  Pepper Julius Panthers       DE 14137500
4  Palmer Carson  Bengals       QB 13800000
5 Manning    Eli   Giants       QB 12916666
```

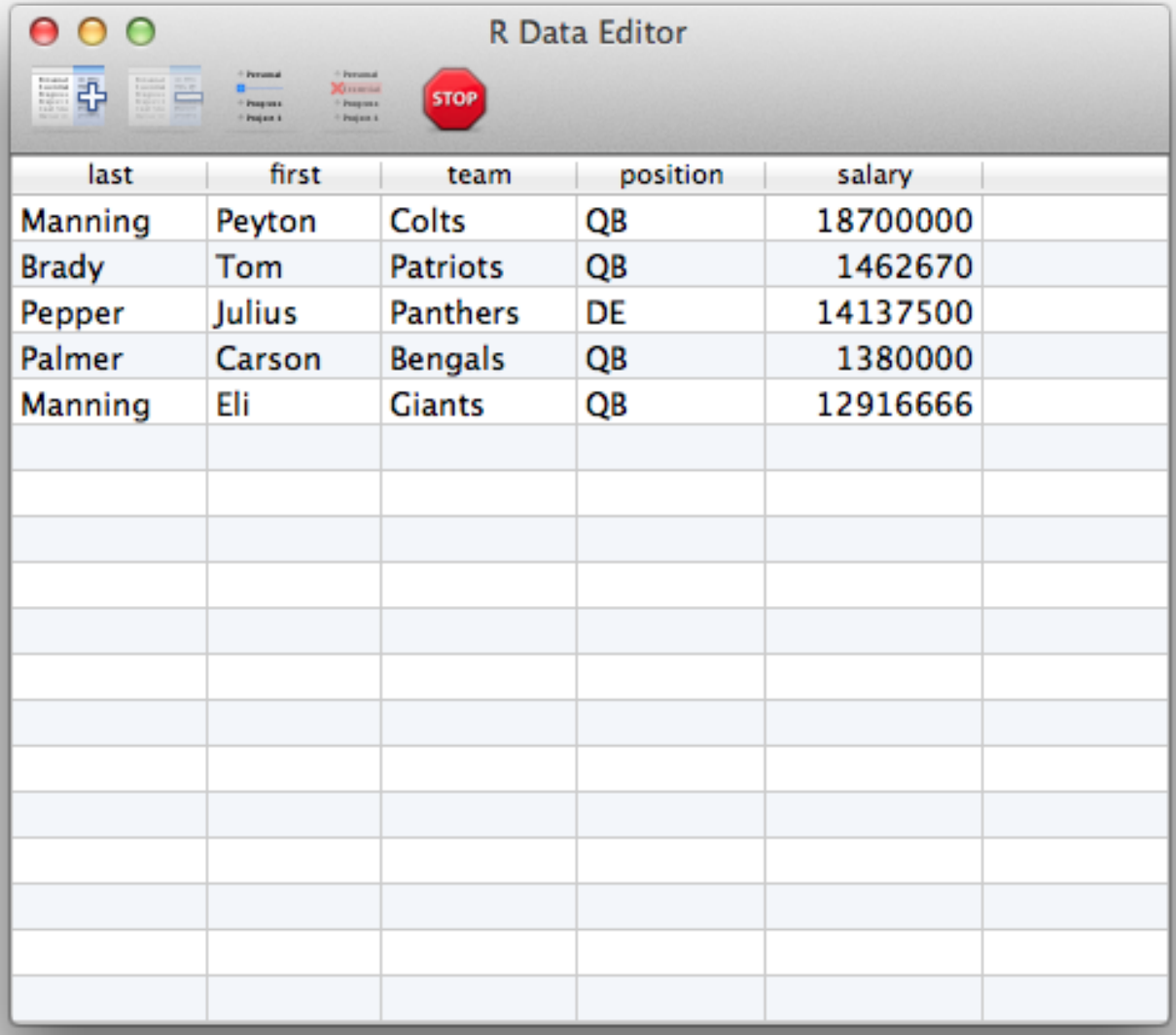
Entering data using individual statements can be awkward for more than a handful of observations.

Luckily R provides a GUI for editing tabular data.

- Use `edit()` to open the data editor.

  ```
  > top5salaries <- edit(top5salaries)
  ```

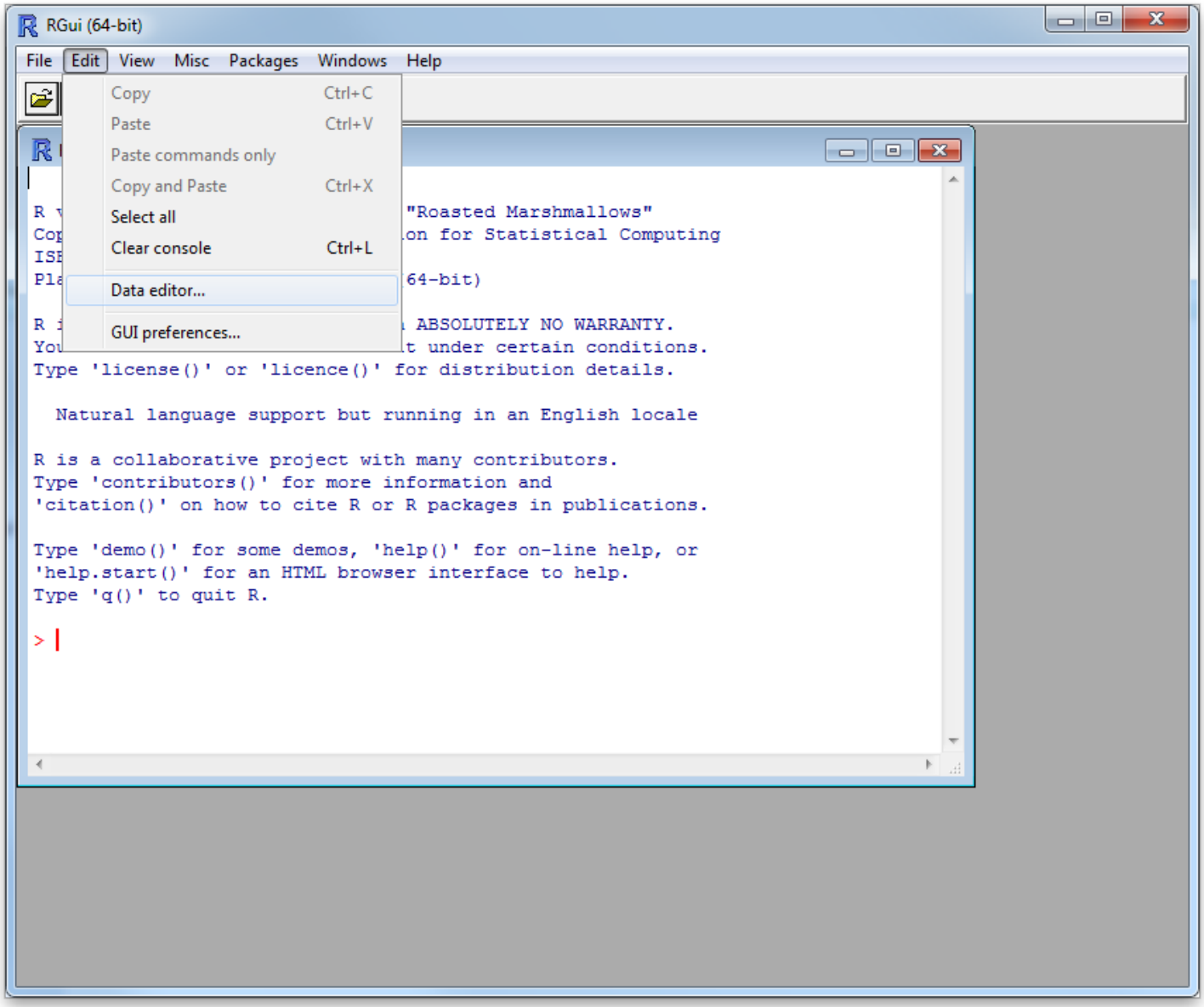
- Notice that you need to assign the output of the `edit()` function to a symbol, otherwise the edits will be lost.
- It is designed for data frames and matrices.
- `edit()` will work on other objects, like vectors, functions, and lists, but it will open a text editor.



last	first	team	position	salary
Manning	Peyton	Colts	QB	18700000
Brady	Tom	Patriots	QB	1462670
Pepper	Julius	Panthers	DE	14137500
Palmer	Carson	Bengals	QB	1380000
Manning	Eli	Giants	QB	12916666

- The versions on Windows and Unix are similar.

- Alternatively, you can use `fix()`, which calls `edit()` on its argument and then assigns the result to the same symbol in the calling environment.
- On Windows, there is a menu item "Data Editor" under the Edit menu that allows you to enter the name of an object into a dialog box.
  - It then calls `fix()` on the object.


RGui (64-bit)
File
Edit
View
Misc
Packages
Windows
Help
Copy Ctrl+C
Paste Ctrl+V
Paste commands only
Copy and Paste Ctrl+X
Select all
Clear console Ctrl+L
Data editor...
GUI preferences...
"Roasted Marshmallows"
on for Statistical Computing
64-bit)
ABSOLUTELY NO WARRANTY.
t under certain conditions.
Type 'license()' or 'licence()' for distribution details.
Natural language support but running in an English locale
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
>

The R data editor can be convenient for inspecting a data frame or matrix, or maybe for editing a couple of values, but I don't recommend using it for doing serious work.

If you have lots of data to enter, use a real spreadsheet, desktop database program, or full-featured editor.

- The R data editor doesn't provide an `Undo` or `Redo` feature;
- it doesn't make it easy to save your work (there is no `Save` button). You need to periodically close the editor to save your work, which is error prone;
- ...

Saving and Loading

The simplest way to save an object is with `save()`.

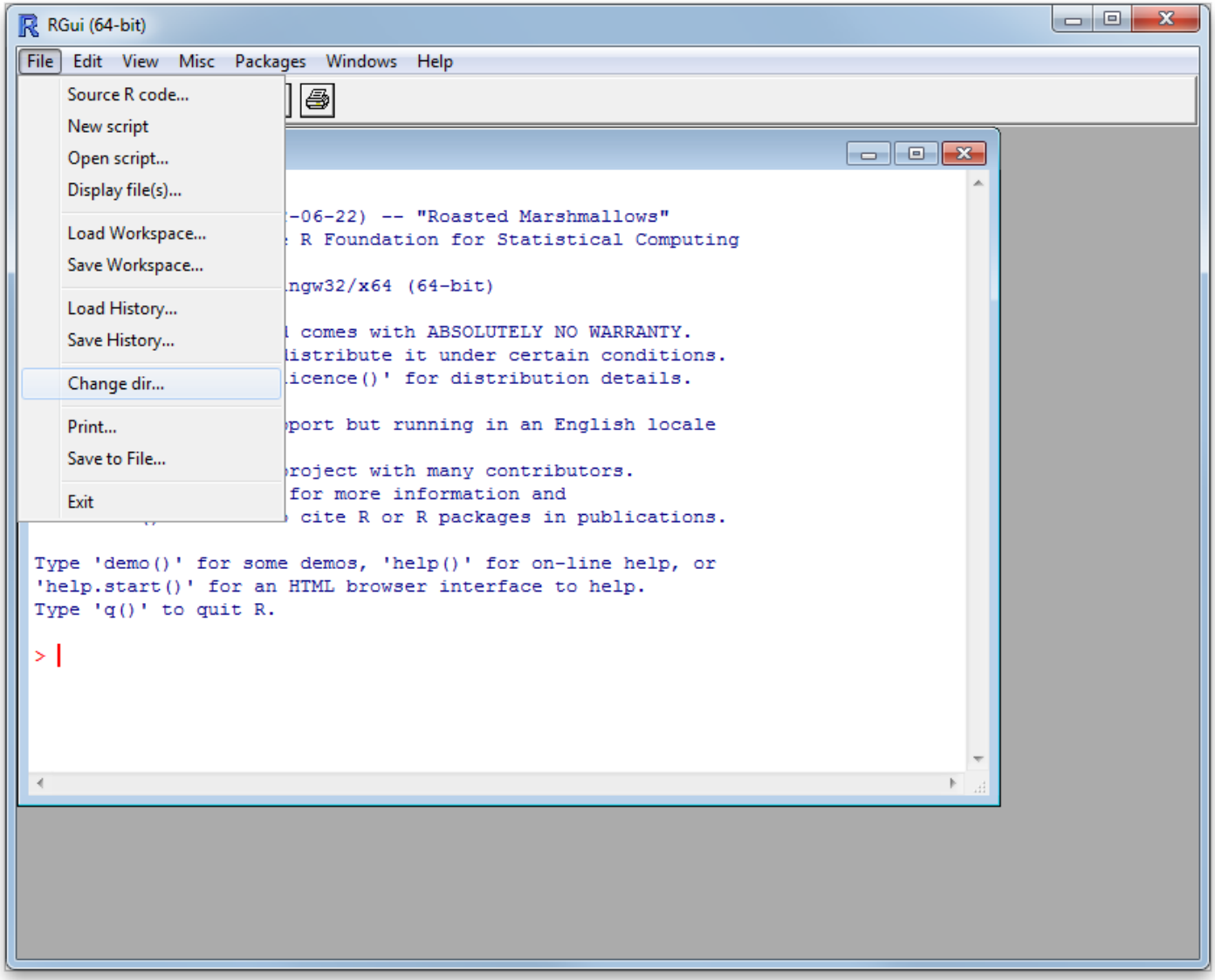
```
> save(top5salaries, file="top5salaries.RData")
```

- In R, file paths are *always* specified with forward slashes ("/"), even in Windows.
- In Windows you'll need `C:/Documents and Settings/...`
- Omitting `~/`, or another *full path*, results in the file being saved to the current working directory (CWD).

```
> getwd()
[1] "~/work-hg/teaching/Rcourse/R"
```

More about paths:

- You can change the CWD with `setwd()`, and the GUIs have pull-down menus, etc., which allow you to browse for the directory you want.
  - In Windows it is in the File menu.
  - In R-Studio it is in the Session menu.
  - In OSX its in the Misc menu.
- In Unixes the CWD will be set as whatever directory you started R in.
- The ".RData" extension is a convention, but not required.


RGui (64-bit)
File Edit View Misc Packages Windows Help
Source R code...
New script
Open script...
Display file(s)...
Load Workspace...
Save Workspace...
Load History...
Save History...
Change dir...
Print...
Save to File...
Exit
-06-22) -- "Roasted Marshmallows"
R Foundation for Statistical Computing
ngw32/x64 (64-bit)
comes with ABSOLUTELY NO WARRANTY.
istribute it under certain conditions.
icence()' for distribution details.
port but running in an English locale
roject with many contributors.
for more information and
cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
>

More about objects being saved:

- You can list multiple objects for saving via `save()`.
- You can save every object in the workspace with `save.image()`.

```
> save.image(file="data.RData")
```

R offers to do this for you automatically when you quit via `q()`.

- It calls `save.image(file=".RData")`, saving the current workspace as a hidden file in the CWD.
- These files have the same structure as those `save()`d.

You can load a saved object back into R with `load()`.

```
> load("top5salaries.RData")
```

- This causes the saved object(s) to be loaded into the current environment, and assigns them the same symbols they had when they were saved.
- Careful: any pre-existing symbols pointing to other objects will be written over (destroyed) without warning.
- Files `save()`d in R will work across platforms when `load()`ed.
- When R starts up it will automatically `load()` the `.RData` file in the CWD if there is one.

Importing Data from External Files

One of the nicest things about R is how easy it is to pull data from other programs.

R can import data from

- text files,
- other statistics software,
- or spreadsheets.

You don't even need a local copy of the file.

- You can specify a file at a URL, and R will fetch it for you over the internet.

R includes a family of functions for importing delimited text files

- where each line contains a record of different variables associated with each observation, which are
- separated by a delimiter, like "whitespace" or commas.
- Each line/record must have the same number of delimiters (variable).

They are all based on the `read.table()` function:

```
read.table(file, header = FALSE, sep = "", ...)
```

For example, suppose you had a file called `top5salaries.csv` containing our salary data, where

- the first row contains the column names (without quotes);
- each (non-numeric) string is encapsulated in quotes;
- and each field is separated by commas.

Then we could load this data into R as follows.

```
> top5salaries <-
+    read.table("top5salaries.csv", header=TRUE,
+               sep=",")
```

- The object returned is a data frame.

`read.table()` has many different options

- see `? read.table`

The most important are

- `sep=`: giving the delimiter, and
- `header=`: specifying whether or not the first line of the file is special.

When loading large files you might find after a long wait that you've misspecified something, like the `sep=` or `header=`.

- A useful technique in this case is to first read a small number of rows with `nrows=20`.

R includes a few other functions that call `read.table()` in convenient ways:

- `read.csv()` and `read.csv2()` for comma-separated files with headers or semi-colon separated ones that might use commas for decimals.
- `read.delim()` and `read.delim2()` for tab-separated files with headers that might use commas for decimals.

In most cases you will find that you can use `read.csv` and `read.delim` unless you are in Europe.

```
> top5salaries <- read.csv("top5salaries.csv")
```

As another example, suppose that you wanted to analyze some historical stock quote data.

- Yahoo! Finance provides this information in an easily downloadable form on its website.
- You can fetch a CSV file from a single URL.

For example, suppose we want to download price info for the S&P 500 index for every month between April 1, 2007 and April 1, 2015, stored on the class web page

`http://bobby.gramacy.com/teaching/asc/sp500.csv`

Conveniently, you can use a URL in place of a filename in R.

```
> url<-"http://bobby.gramacy.com/teaching/asc/sp500.csv"
> sp500 <- read.csv(url)
> sp500[1:5,1:6]
        Date  Open  High   Low Close Adj.Close
1 2007-04-01 26.38 26.39 25.61 25.75     25.75
2 2007-05-01 25.90 26.50 25.85 26.22     26.22
3 2007-06-01 26.25 26.65 25.92 26.07     26.07
4 2007-07-01 26.10 26.25 25.49 25.60     25.60
5 2007-08-01 25.40 25.47 23.25 24.09     24.09
```

There are a couple of other ways to get textual data into R which are handy when the data are not formatted appropriately for `read.table()`.

- E.g., observations in the file might span multiple lines,
- or may not be consistently delimited.

One example is `readLines()`.

- reading each line as a character string, forming a vector of characters.
- By default it facilitates interactive input from the console.

(see `.R` file)

The `scan()` function works similarly, but allows you to read the contents of a file into a specifically defined data structure using the `what=` argument.

I find it most useful in its default configuration

- `what = double(0)`

which allows you to read in an (unstructured) vector of double-precision numbers separated by whitespace.

Exporting data

The "inverse" of `read.table()` is `write.table()`.

- Here is how I created that file we read in earlier.

```
> write.table(top5salaries,
+     file="top5salaries.csv",
+     sep=",", row.names=FALSE)
```

- Similarly, there are wrapper functions `write.csv()` and `write.csv2()` which offer a shorthand.

The `cat()` function is closest analog to the "inverse" of readLines()

- allowing you two write arbitrary characters to a file.

```
> x <- c(7, 11, 24, 28)
> cat("my favorite number is", x[1], "\n",
+     file="favs.txt")
> cat("followed by", x[-1], "\n",
+     file="favs.txt", append=TRUE)
```

- We've already seen how `cat()` can be helpful for printing messages/progress indicators to the screen.

The `write()` function is deceptively useless.

- It is a wrapper around `cat()` which is designed to help write matrices to files.
- But you have to transpose your matrix and specify the number of (pre-transposed) columns.

```
> write(t(top5salaries), file="top5salaries.txt",
+       ncolumns=ncol(top5salaries))
```

- As a result, its both harder to use and less functional than `write.table()`.

Objects/data from other softwares

R can read/write data saved in other languages proprietary/binary formats.

- The `foreign` library supports Stata, SPSS, SAS, Octave, Minitab, Systat and several others.
- The `Rmatlab` package can read and write "MAT" files.
- There are dozens which work with Excel in some way and allow reading and writing of "XLS" files.

Most of data-oriented software programs allow data to be saved in a CSV format, which is often the best way to go.

Interfacing with databases

Many large companies, healthcare providers, and academic institutions keep data in relational databases.

- E.g., SQL variants, DB2, Oracle, Teradata, Sybase, ...

You can always export data from a database to a text file and then import that into R.

- If you plan to export a large amount of data once and then analyze it (once), this is often the best approach.
- However, if you are using R to produce regular reports or a repeat analyses, then it might be better to import data into R directly through a database connection.

To connect directly to a database you will need to install a package, the best of which depend on which you want to connect to and by what method.

- `RODBC`: allows you to fetch data from ODBC (Open DataBase Connectivity) connections, which is a standardized interface.
- `DBI`: which allows connections to databases using native database drivers or JBC drivers. The package is an *abstraction* and requires the installation of additional packages to use the native drivers for each database.
- `TSDBI:` specifically designed for time series data.

Third-party drivers may be required for particular databases.

Lets take DBI as a quick example. DBI is actually a framework and set of packages.

- E.g., `RSQLite` is a DBI package.

```
> install.packages("RSQLite")
> library(RSQLite)
Loading required package: DBI
```

Consider some baseball data stored in an SQLite database.

```
> con <- dbConnect("SQLite", dbname="bb.db")
```

Here are the list of tables available through this database connection.

```
> dbListTables(con)
 [1] "Allstar"                "AllstarFull"
 [3] "Appearances"            "AwardsManagers"
 [5] "AwardsPlayers"          "AwardsShareManagers"
 [7] "AwardsSharePlayers"     "Batting"
 [9] "BattingPost"            "Fielding"
[11] "FieldingOF"             "FieldingPost"
[13] "HOFold"                 "HallOfFame"
[15] "Managers"               "ManagersHalf"
...
```

To find the list of columns for one of the tables, use `dbListFields()`.

```
> dbListFields(con, "AllStar")
[1] "playerID" "yearID"   "lgID"
```

Use `dbGetQuery()` to fetch a data frame with the results.

- E.g., the wins and losses for AL teams in 2008.

```
> wlAL08 <- dbGetQuery(con, paste(
+     "SELECT teamID, W, L FROM Teams",
+      "where yearID=2008 and lgID='AL'"))
```

Lets extract a couple for later use.

```
> batting <- dbGetQuery(con,
+     "SELECT * FROM Batting")
> write.csv(batting, file="batting.csv",
+     row.names=FALSE)
> master <- dbGetQuery(con,
+     "SELECT * FROM Master")
> write.csv(master, file="master.csv",
+     row.names=FALSE)
```

And then close down the connection.

```
> dbDisconnect(con)
[1] TRUE
```

Preparing data

In practice, data is almost never stored in the right form for analysis.

- Even when it is, there are can be "surprises".
- It can take a lot of work to pull together a usable data set.
- R has many helpful tools for the task.

We'll begin with combining data stored in separate objects.

- E.g., combining batting stats with player age; player bio data is usually stored separately from performance data.

... but first some basics.

The simplest combine is a `paste()` of two sets of character strings.

```
> x <- c("a", "b", "c", "d", "e")
> y <- c("A", "B", "C", "D", "E")
> paste(x, y, sep="-")
[1] "a-A" "b-B" "c-C" "d-D" "e-E"
> paste(x, y, sep="-", collapse=":")
[1] "a-A:b-B:c-C:d-D:e-E"
```

`rbind()` and `cbind()` help bind together multiple data frames, matrices, or vectors

- by row or column.

Consider adding year and rank info to our salary data.

```
> year <- rep(2008, 5)
> rank <- 1:5
> morecols <- data.frame(year, rank)
> cbind(top5salaries, morecols)
     last  first     team position   salary year rank
1 Manning Peyton    Colts       QB 18700000 2008    1
2   Brady    Tom Patriots       QB 14626700 2008    2
3  Pepper Julius Panthers       DE 14137500 2008    3
4  Palmer Carson  Bengals       QB 13800000 2008    4
5 Manning    Eli   Giants       QB 12916666 2008    5
```

Consider adding a few more players. (see .R file)

```
> rbind(top5salaries, next3)
      last  first     team position   salary
1  Manning Peyton    Colts       QB 18700000
2    Brady    Tom Patriots       QB 14626700
3   Pepper Julius Panthers       DE 14137500
4   Palmer Carson  Bengals       QB 13800000
5  Manning    Eli   Giants       QB 12916666
6    Favre   Bret  Packers       QB 12800000
7   Bailey  Champ  Broncos       CB 12600000
8 Harrison Marvin    Colts       WR 12000000
```

Lets extend the stock quotes example.

Suppose we wanted a single data set with stock quotes for multiple securities,

- e.g., the 30 in the DJIA
- all bound together into a single data frame.

We'll write a functions that

- assemble the correct URL to pull quotes for particular stocks over the last 365 days,
- and another that combines them together, iteratively calling the first function.

The function `get.quotes()` is provided in the accompanying `.R` file.

```
> get.multiple.quotes <-
+    function(tkrs, crumb, from=(Sys.Date()-365),
+             to=(Sys.Date()), interval="d")
+   {
+     tmp <- NULL
+     for (tkr in tkrs) {
+       if (is.null(tmp))
+         tmp <- get.quotes(tkr,from,to,interval)
+       else tmp <- rbind(tmp,
+                 get.quotes(tkr,from,to,interval))
+     }
+     tmp
+   }
```

```
> ## get the tickers
> dow30.tickers <-
+   c("MMM", "AXP", "AAPL", "BA", "CAT", "CVX","CSCO",
+     "KO", "DIS", "XOM", "GE", "GS", "HD", "IBM",
+     "INTC", "JNJ", "JPM", "MCD", "MRK", "MSFT", "NKE",
+     "PFE", "PG", "TRV", "UTX", "UNH", "VZ", "V", "WMT")
> Sys.Date()
[1] "2017-09-20"
> ## you need to get the crumb from the browser
> dow30 <- get.multiple.quotes(dow30.tickers,
+     crumb="KBKzdmRrCb6")
```

We'll return to this example shortly.

Merging data

Returning to the baseball data, suppose we wanted to `merge()` the `batting` and `master` databases to combine performance and bio data.

```
> rbind(dim(batting), dim(master))
      [,1] [,2]
[1,] 91457   24
[2,] 17264   33
> both <- merge(batting, master)
> dim(both)
[1] 91457    56
```

By default, `merge()` uses common variables between the two data frames as the merge keys.

- So in this case we did not have to specify any more arguments to `merge()`.
- By default, `merge()` is equivalent to a NATURAL JOIN in SQL.
- It can also do an INNER JOIN,
- an OUTER or FULL join,
- or the full Cartesian product of the two data sets.

Transformations

Sometimes there will be some variables in your source data that aren't quite right.

One of the most convenient ways to redefine a variable in a data frame is to use the assignment operator.

For example, suppose we wanted to change the type of a variable in the `dow30` data frame that we created above.

When `read.csv()` imported the data, it interpreted the `$Date` field as a character string and converted it into a factor

```
> class(dow30$Date)
[1] "factor"
```

It would be more conveient (e.g., for ordering purposes) to have that part of the data as a `Date` object.

```
> dow30$Date <- as.Date(dow30$Date)
> class(dow30$Date)
[1] "Date"
```

- Luckily, Yahoo! Finance gives dates in a standard format.

No suppose we wanted to define a new midpoint variable that is the average of the high and low price.

```
> dow30$Mid <- (dow30$High + dow30$Low)/2
> names(dow30)
[1] "symbol"    "Date"      "Open"
[4] "High"      "Low"       "Close"
[7] "Volume"    "Adj.Close" "Mid"
```

A convenient way of changing variables in a data frame is via `transform()`.

Formally, its defined as

`transform('_data', ...)`

i.e., without many *named* arguments. You specify

- a data frame (as the first argument)
- and a set of expressions that use variables within the data frame.

`transform()` applies each expression to the data frame and then returns the final data frame.

E.g., suppose we wanted to perform the two transformations we just did, above.

```
> dow30t <- get.multiple.quotes(dow30.tickers)
> dow30t <- transform(dow30t, Date=as.Date(Date),
+                     mid=(High+Low)/2)
> names(dow30t)
[1] "symbol"    "Date"      "Open"
[4] "High"      "Low"       "Close"
[7] "Volume"    "Adj.Close" "mid"
> class(dow30t$Date)
[1] "Date"
```

Apply

The `apply` family of methods are useful for transforming data too. We've seen some examples already with

- `sapply()` for vectors;
- `apply()` for arrays and matrices;
- `lapply()` works on vectors, lists, or data frames and returns a list;
- `mapply()`: "multivariate" version of `sapply()`;
- `tapply()` and `by()` specifically for collapsing info/summarizing.

`apply()` is the most useful; it even works with $>$ 2d arrays ..

```
> x <- 1:27
> dim(x) <- c(3,3,3)
> apply(x, 1, paste, collapse=",")
[1] "1,4,7,10,13,16,19,22,25"
[2] "2,5,8,11,14,17,20,23,26"
[3] "3,6,9,12,15,18,21,24,27"
> apply(x, 2, paste, collapse=",")
[1] "1,2,3,10,11,12,19,20,21"
[2] "4,5,6,13,14,15,22,23,24"
[3] "7,8,9,16,17,18,25,26,27"
> apply(x, 3, paste, collapse=",")
[1] "1,2,3,4,5,6,7,8,9"
[2] "10,11,12,13,14,15,16,17,18"
[3] "19,20,21,22,23,24,25,26,27"
```

You can even get complicated and specify the `MARGIN=` (the second argument) to be more than one column.

```
> apply(x, 1:2, paste, collapse=",")
     [,1]       [,2]       [,3]
[1,] "1,10,19" "4,13,22" "7,16,25"
[2,] "2,11,20" "5,14,23" "8,17,26"
[3,] "3,12,21" "6,15,24" "9,18,27"
```

- For each value of i between 1 and 3,
- and each value of j between 1 and 3,
- calculate the `FUN=` (the function) of

`x[i][j][1], x[i][j][2], x[i][j][3]`

Binning Data and Subsets

Another common data transformation is to group a set of observations into bins based on the value of a specific variable.

- E.g., converting a daily time series into a monthly one.

Many of the functions we've seen already are useful when binning,

- e.g., `transform()`, `seq()`, and the bracket notation `[]`.

We'll look at some new ones too and see them all in action.

The `cut()` function is useful for taking a continuous variable and splitting it into discrete pieces.

`cut(`*x*, *breaks*, *...*`)`

- taking a vector (`x=`) as input and returning a factor;
- each level of the factor corresponds to an interval of values in the input vector.
- `breaks=` either specifies the number of groups, or gives the cut points for the groups.

For example, suppose we wanted to count the number of baseball players with averages in certain ranges.

```
> bat08 <- read.csv("../data/bat08.csv")
> bat08 <- transform(bat08, AVG=H/AB)
> bat08.100AB <- subset(bat08, AB>100)
> ## same as bat08[bat08$AB > 100,]
> av08bins <- cut(bat08.100AB$AVG, breaks=10)
```

```
> table(av08bins)
av08bins
(0.137,0.163] (0.163,0.189] (0.189,0.215]
            4             6            24
 (0.215,0.24]  (0.24,0.266] (0.266,0.292]
           67           121           132
(0.292,0.318] (0.318,0.344]  (0.344,0.37]
           70            11             5
 (0.37,0.396]
            2
```

- Here, `table()` is simply counting the number of that have each possible categorical value.

As another example of `table()` consider using it to count the number of left and right-handed batters, and switch-hitters.

```
> table(bat08$bats)
  B   L   R
118 401 865
```

Or, a 2-d table including throwing hand.

```
> table(bat08[, c("throws", "bats")])
      bats
throws   B   L   R
     L  10 240  25
     R 108 161 840
```

`tabulate()` helps count the number of non-categorical observations that take on each possible value.

E.g., suppose we wanted a count of the number of players who hit $0, 1, 2, \ldots$ home runs.

```
> HRcnts <- tabulate(bat08$HR)
> names(HRcnts) <- 0:(length(HRcnts)-1)
> HRcnts
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
92 63 45 20 15 26 23 21 22 15 15 18 12 10 12  4
16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
 9  3  3 13  9  7 10  4  8  2  5  2  4  0  1  6
...
```

Often it is desirable to take a random sample of a data set.

- Sometimes you have too much data (for statistical or performance reasons).
- Other times you might want to split the data into different parts for modeling (e.g., into training, testing and validation sets).

One of the simplest ways to extract a random sample is via `sample()`, which returns a sample of the elements of a vector.

```
> sample(1:10, 9)
[1] 10  4  1  9  6  8  5  2  3
> sample(1:10, 9, replace=TRUE)
[1] 6 6 7 4 2 8 6 2 8
```

To take a random sample of the observations in a data frame,

- use `sample()` to create a random sample of row numbers,
- then select these row numbers using an index operator.

```
> bat08[sample(1:nrow(bat08), 5),1:5]
     nameLast nameFirst weight height bats
914   Britton     Chris    278     75    R
753    Maholm      Paul    225     74    L
629 Hennessey      Brad    185     74    R
797    Iguchi  Tadahito    185     70    R
242  Matthews      Gary    210     75    B
```

What about more complicated random subsets? E.g., if you wanted to randomly select statistics for three teams.

```
> bat08$teamID <- as.factor(bat08$teamID)
> teams <- levels(bat08$teamID)
> bat08.3t <- bat08[bat08$teamID %in%
+                   sample(teams, 3),]
> summary(bat08.3t$teamID)
ARI ATL BAL BOS CHA CHN CIN CLE COL DET FLO HOU
  0   0   0  47   0   0   0   0   0   0   0   0
KCA LAA LAN MIL MIN NYA NYN OAK PHI PIT SDN SEA
  0   0   0   0   0  51  50   0   0   0   0   0
SFN SLN TBA TEX TOR WAS
  0   0   0   0   0   0
```

`tapply()` and `by()` are useful functions for summarizing the columns of a matrix or data frame.

- They work like other `apply()` functions.

`aggregate()` is another, which works on data frames and time series objects.

- E.g., to summarize batting stats by team.

```
> cols <- c("AB", "H", "BB", "X2B", "X3B", "HR")
> aggregate(bat08[,cols], by=list(bat08$teamID),
+           sum)
   Group.1   AB    H  BB X2B X3B  HR
1      ARI 5409 1355 587 318  47 159
2      ATL 5604 1514 618 316  33 130
...
```

Finding and removing duplicates

Duplicated data can sometimes cause problems.

- R provides some useful functions for detecting duplicate values.

E.g., suppose you accidentally included one stock ticker twice.

```
> tickers2 <- c("GE", "GOOG", "AAPL",
+     "AXP", "GS", "GE")
> quotes2 <- get.multiple.quotes(tickers2,
+     from=as.Date("2009-01-01"),
+     to=as.Date("2009-03-31"), interval="m")
```

`duplicated()` returns a logical vector showing which elements (entries or rows) are duplicates.

```
> duplicated(quotes2)
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [8] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[15] FALSE  TRUE  TRUE  TRUE
> uquotes2 <- quotes2[!duplicated(quotes2),]
> dim(uquotes2)
[1] 15  8
```

Here is another way to do the same thing.

```
> uquotes2 <- unique(quotes2)
```

Sorting

Sorting vectors is straightforward in R.

```
> w <- c(5,4,7,2,7,1)
> sort(w)
[1] 1 2 4 5 7 7
> sort(w, decreasing=TRUE)
[1] 7 7 5 4 2 1
```

- By default, any `NA` items are not shown.
- By adjusting the `NA.last` argument you can make them appear first or last.

Sorting a data frame is a little more complicated.

- First use `order()` to obtain the sorted order of one of the indices.
- Then reorder the rows of the data frame according to that order.

`order()` works like `sort()` but gives you the permutation of the indices (to sort them) rather than actually sorting them.

```
> o <- order(w)
> o
[1] 6 4 2 1 3 5
> w[o]
[1] 1 2 4 5 7 7
```

Suppose we created the following data frame from the vector `w` and a second vector `u`.

```
> u <- c("pig", "cow", "duck", "horse", "rat",
+     "moose")
> v <- data.frame(w, u)
> v
  w     u
1 5   pig
2 4   cow
3 7  duck
4 2 horse
5 7   rat
6 1 moose
```

We could sort the data frame `v` by `w` using:

```
> v[order(v$w),]
  w     u
6 1 moose
4 2 horse
2 4   cow
1 5   pig
3 7  duck
5 7   rat
```

- You can pass multiple vectors to `order()` in order to sort via multiple keys. (See `.R` file for example.)