

Stat 260, Lecture 5, Reading Data

Brad McNeney

Load packages

```
library(tidyverse)  
library(nycflights13)
```

Reading

- ▶ Workflow: scripts: Chapter 4 of printed text, Chapter 6 of online text
- ▶ Introduction to data wrangling: Part II introduction of printed text, Chapter 9 of online text
- ▶ Tibbles: Chapter 7 of printed text, Chapter 10 of online text
- ▶ Reading data with readr: Chapter 8 of printed text, Chapter 11 of online text
- ▶ Data import (readr/tidyr) cheatsheet at [<https://github.com/rstudio/cheatsheets/raw/master/data-import.pdf>]

Tibbles

- ▶ In base R, the data structure used to hold data sets is the data frame.
- ▶ We can make a data frame from vectors as follows:

```
dd <- data.frame(x=c(NA,10,1),y=c("one","two","three"))  
dd
```

```
##      x      y  
## 1 NA    one  
## 2 10    two  
## 3  1 three
```

- ▶ The tidyverse authors find the default behaviour of data frames to be odd, and so implemented an improvement called tibbles:

```
tt <- tibble(x=c(NA,10,1),y=c("one","two","three"))  
tt
```

```
## # A tibble: 3 x 2  
##       x y  
##   <dbl> <chr>  
## 1    NA one  
## 2    10 two  
## 3     1 three
```

data frames to tibbles and back

- ▶ data frames can be coerced to tibbles and *vice versa*.

```
as_tibble(dd)
```

```
## # A tibble: 3 x 2
##       x y
##   <dbl> <fct>
## 1     NA one
## 2     10 two
## 3      1 three
```

```
as.data.frame(tt)
```

```
##      x      y
## 1 NA    one
## 2 10    two
## 3 1    three
```

tibble printing

- ▶ One difference between data frames and tibbles is how they are printed.
- ▶ Printing a data frame: all rows and columns, up to your R session's `max.print`.
- ▶ Printing a tibble: the first 10 rows, as many columns as fit the screen, and the column data types.

```
flights
```

```
## # A tibble: 336,776 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>   <int>         <int>         <dbl>   <int>
## 1  2013     1     1     517             515           2     830
## 2  2013     1     1     533             529           4     850
## 3  2013     1     1     542             540           2     923
## 4  2013     1     1     544             545          -1    1004
## 5  2013     1     1     554             600          -6     812
## 6  2013     1     1     554             558          -4     740
## 7  2013     1     1     555             600          -5     913
## 8  2013     1     1     557             600          -3     709
## 9  2013     1     1     557             600          -3     838
## 10 2013     1     1     558             600          -2     753
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dtm>
```

Control printing of tibbles

- ▶ To see all rows/columns of a tibble, best to `View()` it.
- ▶ But you can also print all rows and columns by setting `options(dplyr.print_min=Inf)` and `options(tibble.width=Inf)`.

Extracting columns as vectors

- ▶ Use the basic tools `$` and `[[` to extract a variable from a tibble or data frame:

```
dd$x
```

```
## [1] NA 10 1
```

```
tt$x
```

```
## [1] NA 10 1
```

```
dd[["x"]]
```

```
## [1] NA 10 1
```

```
tt[["x"]]
```

```
## [1] NA 10 1
```

Subsetting: columns

- ▶ Using `select()` is the preferred method to subset columns of a data frame or tibble, but we can also use the more basic tool `[]`; e.g.,

```
tt[, "x"]
```

```
## # A tibble: 3 x 1
##       x
##   <dbl>
## 1    NA
## 2    10
## 3     1
tt[, c("x", "y")]
```

```
## # A tibble: 3 x 2
##       x y
##   <dbl> <chr>
## 1    NA one
## 2    10 two
## 3     1 three
dd[, "x"] # returns a vector
```

```
## [1] NA 10  1
dd[, c("x", "y")]
```

```
##       x       y
## 1 NA    one
## 2 10    two
## 3 1    three
```

Subsetting: rows

- ▶ Using `filter()` is the preferred method to extract rows of a data frame or tibble, but we can also use `[`.

```
tt[2,]
```

```
## # A tibble: 1 x 2
##       x y
##   <dbl> <chr>
## 1    10 two
```

```
tt[1:2,]
```

```
## # A tibble: 2 x 2
##       x y
##   <dbl> <chr>
## 1    NA one
## 2    10 two
```

```
dd[2,] # returns a vector
```

```
##       x y
## 2 10 two
dd[1:2,]
```

```
##       x y
## 1  NA one
## 2 10 two
```

Exercise

- ▶ Create a data frame `myd` and tibble `myt` that each have columns named `cat`, `dog` and `mouse`. Each column should be of length three, but the values in each column are up to you.
- ▶ What do `names(myd)` and `names(myt)` return?
- ▶ Create the variable `a1 <- c("cat","dog","bird","fish")` and the variable `a2 <- c("cat","tiger")`. We can combine logicals with `[]` to subset. What do the following return?
 - ▶ `myd[,names(myd) %in% a1]`
 - ▶ `myd[,names(myd) %in% a2]`
 - ▶ `myd[,names(myt) %in% a1]`
 - ▶ `myd[,names(myt) %in% a2]`

Importing data

- ▶ We read in the HIV prevalence data with the base R function `read.csv()`, which returned a data frame.
- ▶ We will now discuss the tidyverse equivalent, `read_csv()`, which returns a tibble.

```
hiv <- read_csv("../Labs/HIVprev.csv")
```

```
## Parsed with column specification:
## cols(
##   Country = col_character(),
##   year = col_double(),
##   prevalence = col_double()
## )
```

Why use `read_csv()` instead of `read.csv()`?

- ▶ `read_csv()` reports how each column of the CSV file was “parsed” (more on this later),
- ▶ returns a tibble,
- ▶ uses `stringsAsFactors = FALSE` as the default, (recall `hiv <- read.csv("../Labs/HIVprev.csv", stringsAsFactors = FALSE)`)
- ▶ is faster, and
- ▶ is more consistent across operating systems.

Other read_ functions

- ▶ CSV stands for comma-separated files, aka comma-delimited files
- ▶ `read_csv()` reads semicolon-delimited files,
- ▶ `read_tsv()` reads tab-delimited files,
- ▶ `read_delim()` reads files with user-specified delimiter.
- ▶ Exercise: A file called “chicken.C” contains the following data on two chickens, with IDs 22 and 33, who laid 2 and 1 eggs, respectively. (Reference: <https://isotropic.org/papers/chicken.pdf>) How would you read this data file into R?

IDCeggs

22C2

33C1

Skip and comments

- ▶ Some files contain a header that describes the data, aka meta-data, that we should skip when reading.
- ▶ Some files include comments that start with common characters, such as “#”.
- ▶ Example file

```
This is a header  
that you should skip  
# this is a comment  
A,B,C  
1,2,3  
4,5,6 # another comment
```



```
read_csv("lec05exfile.csv", skip=2, comment="#")
```

```
## Parsed with column specification:
```

```
## cols(
```

```
##   A = col_double(),
```

```
##   B = col_double(),
```

```
##   C = col_date(format = "")
```

```
## )
```

```
## # A tibble: 2 x 3
```

```
##       A      B C
```

```
##   <dbl> <dbl> <date>
```

```
## 1     1    2.2 1999-05-10
```

```
## 2     4    5.5 2001-04-04
```

Parsing a vector

- ▶ `read_csv()` returns a message that described how each column of the input file was parsed.
- ▶ Parsing a file depends on the `parse_*` functions, such as `parse_number()`, that parse vectors.
- ▶ The `parse_*` functions take a vector of character strings as input and return a vector of a given mode, handling missing values as specified by the user.

```
parse_number(c("$10.55", "33%", "Number is 44", "."), na=".")
```

```
## [1] 10.55 33.00 44.00    NA
```

- ▶ The parse functions are designed to handle data formats and character sets from around the world.
- ▶ In this course we assume North American data formats and character set.
- ▶ See the text if you need other formats.

Other parsing functions

- ▶ `parse_logical()`, `parse_integer()`, `parse_double()`, `parse_character()`, `parse_factor()`, `parse_datetime()`, `parse_date()` and `parse_time()`.
- ▶ Use the `str()` function to see the mode of an object:

```
str(parse_logical(c("TRUE", "FALSE")))
```

```
## logi [1:2] TRUE FALSE
```

```
str(parse_logical(c("1", "0")))
```

```
## logi [1:2] TRUE FALSE
```

```
str(parse_integer(c("1", "0")))
```

```
## int [1:2] 1 0
```

```
str(parse_double(c("1", "0")))
```

```
## num [1:2] 1 0
```

```
str(parse_factor(c("1", "0")))
```

```
## Factor w/ 2 levels "1","0": 1 2
```

Dates and times

- ▶ These parsers have default formats for dates and times, but your best bet is to specify the format yourself.
- ▶ The formatting rules are described in `help(strptime)`.

```
dd <- c("05/14/1966/12/34/56", "04/02/2002/07/43/00", "08/17/2005/07/22/00", "08/1  
dd <- parse_datetime(dd, format = "%m/%d/%Y/%H/%M/%S")  
str(dd)
```

```
## POSIXct[1:4], format: "1966-05-14 12:34:56" "2002-04-02 07:43:00" ...  
mean(dd)
```

```
## [1] "1995-09-18 22:59:59 UTC"  
diff(dd)
```

```
## Time differences in days  
## [1] 13106.797 1232.985 1091.374
```

Parsing files

- ▶ `read_csv()` and other read functions guess at the format of each column. Sometimes this works, sometimes not.
- ▶ You can read about how these functions guess in the text.
- ▶ Here we'll focus on manually specifying the format.

```
dat <- read_csv("lec05exfile.csv", skip=2, comment="#")
```

```
## Parsed with column specification:
## cols(
##   A = col_double(),
##   B = col_double(),
##   C = col_date(format = "")
## )
```

- ▶ Cut-and-paste the guess and replace parsers as necessary

```
dat <- read_csv("lec05exfile.csv", skip=2, comment="#",  
               col_types=cols(  
                 A = col_integer(),  
                 B = col_double(),  
                 C = col_date(format = "%Y-%m-%d")  
               )  
)  
str(dat$A)
```

```
## int [1:2] 1 4
```

- ▶ For reproducibility your R scripts should have a manual specification of the parsing of each column, rather than relying on guesses that can change as your data changes.

Exercise

- Copy the following data to a file and read it in to R. Specify the column types yourself, based on the descriptions in the header of the file. Hint: read about `col_factor()`.

```
# Variable fert is a factor that records the  
# type of fertilizer used in the experiment,  
# date records the date and time of the experiment  
# and yield is the yield of corn.  
fert,date,yield  
F1A2,2018/04/01/12/30,22.56  
F1A1,2018/04/02/12/00,26.06  
F2A2,2018/04/01/12/45,32.03  
F2A1,2018/04/02/12/00,33.21
```