Data Management in R

Andrés L. Parrado, Krishanu Chakraborty

February 27, 2019

Section 1

To start off

Disclaimer

- The J-PAL MIT Micromasters - 102x

Disclaimer

- The J-PAL MIT Micromasters - 102x
- Creative Commons Attribution-NonCommercial-NoDerivs 3.0 License

Disclaimer

- The J-PAL MIT Micromasters - 102x
- Creative Commons Attribution-NonCommercial-NoDerivs 3.0 License
- R for Data Science Heavily borrowed from here.

Disclaimer

- The J-PAL MIT Micromasters - 102x
- Creative Commons Attribution-NonCommercial-NoDerivs 3.0 License
- R for Data Science Heavily borrowed from here.
- The World Wide Web

Section 2

Data Input - Baby steps

Overview

- R is flexible and extensible. As a result there are many ways to approach the tasks these materials address. By necessity we've choosen only one in each case as a starting point.

Overview

- R is flexible and extensible. As a result there are many ways to approach the tasks these materials address. By necessity we've choosen only one in each case as a starting point.
- We have aimed the example code to be clear, idiomatic, and safe, but unoptimized. Writing code that runs as quickly as possible is beyond the scope of these materials.

Quick recap - getting help

- R's built-in documentation can be accessed in several ways. If you know the name of the function you need help with, you can use `?` or `help()`. For example: `?library`; `help("library")`.

Quick recap - getting help

- R's built-in documentation can be accessed in several ways. If you know the name of the function you need help with, you can use `?` or `help()`. For example: `?library`; `help("library")`.
- For interactive help in your browser, use `help.start()`.

Quick recap - getting help

- R's built-in documentation can be accessed in several ways. If you know the name of the function you need help with, you can use `?` or `help()`. For example: `?library`; `help("library")`.
- For interactive help in your browser, use `help.start()`.
- Finally, to search the documentation for a topic, you can use `??`, as in `??library`.

Preliminaries - Installing packages

Let's install packages that will help you get started quickly. The function for this is `install.packages()`.

```
install.packages("tidyverse")
install.packages("dplyr")
install.packages("reshape2")
```

Preliminaries - Package availability

After installation, make the functions in a package easily available with the `library()` function:

```
library("tidyverse")
library("dplyr")
library("reshape2")
```

Reading data into R

- Your approach will depend on the format of the data. We will practice three examples of loading data in common formats, and then point to good solutions for other formats.

Reading data into R

- Your approach will depend on the format of the data. We will practice three examples of loading data in common formats, and then point to good solutions for other formats.
- Please start an R Script.

R dataset (Rdata/Rda)

- Like Stata, R has native data formats. The file extensions `.Rda` and `.Rdata` indicate R's native dataset. (Less frequently, there is also `.Rds`.)

R dataset (Rdata/Rda)

- Like Stata, R has native data formats. The file extensions `.Rda` and `.Rdata` indicate R's native dataset. (Less frequently, there is also `.Rds`.)
- The `load()` function can handle these formats. For example, we've included with these materials an `Rdata` file called `airquality.Rdata`.

Example : Rdata

```
# Create airquality.Rdata for use in the example
data(airquality, package = "datasets")
save(airquality, file = "airquality.Rdata")
# Clear the workspace
rm(list = ls())
```

```
# Read the file into R
load("airquality.Rdata")

# View the objects in our environment
ls()
## [1] "airquality"
```

Notes

- `load()` reads files of `.Rda` and `.Rdata` format and places them in your environment.

Notes

- `load()` reads files of `.Rda` and `.Rdata` format and places them in your environment.
- The example also used `ls()`: It shows the objects in our environment.

Notes

- `load()` reads files of `.Rda` and `.Rdata` format and places them in your environment.
- The example also used `ls()`: It shows the objects in our environment.
- Did trying to run the example code give you an error? We assumed that `airquality.Rdata` was in the current working directory, but perhaps this wasn't true.

Troubleshooting:

```
# Check the current working directory
getwd()
# Output omitted

# Show the files in the current working directory
list.files()
# Output omitted
```

Troubleshooting:

You can change the working directory with `setwd()`.

For example, we could make it any directory in the user's home:

```
# Set the current working directory

setwd("~/Dropbox (IDinsight)/#r_evolution/Intro to R/Session 1 - data management")
```

Stata dataset (dta)

- If you need to transfer data from Stata to R, the `read_dta()` function is a good choice.

Stata dataset (dta)

- If you need to transfer data from Stata to R, the `read_dta()` function is a good choice.
- Warning: unlike `load()`, which will always be available to you when using R, `read_dta()` is in the tidyverse's haven package. The example below will run successfully after you:

Stata dataset (dta)

- If you need to transfer data from Stata to R, the `read_dta()` function is a good choice.
- Warning: unlike `load()`, which will always be available to you when using R, `read_dta()` is in the tidyverse's haven package. The example below will run successfully after you:

1. Call `install.packages("tidyverse")` (once) to install the tidyverse packages

Stata dataset (dta)

- If you need to transfer data from Stata to R, the `read_dta()` function is a good choice.
- Warning: unlike `load()`, which will always be available to you when using R, `read_dta()` is in the tidyverse's haven package. The example below will run successfully after you:

1. Call `install.packages("tidyverse")` (once) to install the tidyverse packages
2. Call `library("haven")` to make them available (once per session).

Create data

```
# Create airquality.dta for use in the example
data(airquality, package = "datasets")
# Fix a Stata-invalid column name
airquality <- dplyr::rename(airquality, Solar_R = Solar.R)
haven::write_dta(airquality, "airquality.dta")
```

Example:

```
library(haven)
airquality_stata <- read_dta("airquality.dta")
head(airquality_stata)
## # A tibble: 6 x 6
##   Ozone Solar_R  Wind  Temp Month   Day
##   <dbl>   <dbl> <dbl> <dbl> <dbl> <dbl>
## 1    41     190   7.4    67     5     1
## 2    36     118   8      72     5     2
## 3    12     149  12.6    74     5     3
## 4    18     313  11.5    62     5     4
## 5    NA      NA  14.3    56     5     5
## 6    28      NA  14.9    66     5     6
```

Summary:

- Whereas `load()` puts saved objects in our environment, `read_dta` returns the tabular dataset from Stata.

Summary:

- Whereas `load()` puts saved objects in our environment, `read_dta` returns the tabular dataset from Stata.
- We used R's assignment operator <- to assign the return value of `read_dta()` to an object that we decided to call `airquality_stata`.

Summary:

- Whereas `load()` puts saved objects in our environment, `read_dta` returns the tabular dataset from Stata.
- We used R's assignment operator <- to assign the return value of `read_dta()` to an object that we decided to call `airquality_stata`.
- Then we looked at the initial rows of the table with `head()`.

Comma-separated (csv)

Unprocessed tabular data is often in comma-separated (csv) format. A good function for reading it is `read_csv()`, which is in the tidyverse's `readr` package.

```
# Create airquality.dta for use in the example
data(airquality, package = "datasets")
readr::write_csv(airquality, "airquality.csv")
```

Example:

```
library(readr)
airquality_csv <- read_csv("airquality.csv")
## Parsed with column specification:
## cols(
##   Ozone = col_double(),
##   Solar.R = col_double(),
##   Wind = col_double(),
##   Temp = col_double(),
##   Month = col_double(),
##   Day = col_double()
## )
```

Example:

```
head(airquality_csv)
## # A tibble: 6 x 6
##   Ozone Solar.R  Wind  Temp Month   Day
##   <dbl>   <dbl> <dbl> <dbl> <dbl> <dbl>
## 1    41     190   7.4    67     5     1
## 2    36     118   8      72     5     2
## 3    12     149  12.6    74     5     3
## 4    18     313  11.5    62     5     4
## 5    NA      NA  14.3    56     5     5
## 6    28      NA  14.9    66     5     6
```

Note:

- The precise format of CSV files varies. In the future, you may need to use additional arguments to `read_csv()` For example, to ignore the first row in a CSV file, we could use argument `skip = 1`. See `help("read_csv", package = "readr")`.

Note:

- The precise format of CSV files varies. In the future, you may need to use additional arguments to `read_csv()` For example, to ignore the first row in a CSV file, we could use argument `skip = 1`. See `help("read_csv", package = "readr")`.
- `read_csv()` sent some output to the console relating to the type of data in each column in `airquality.csv`. We'll return to the subject of data types on a sunnier day.

Other formats:

To read data that arrives in some other format we haven't discussed, these functions may be useful.

- Whitespace-separated text - `read_table()`

Other formats:

To read data that arrives in some other format we haven't discussed, these functions may be useful.

- Whitespace-separated text - `read_table()`
- Tab-separated (TSV) text - `read_tsv()`

Other formats:

To read data that arrives in some other format we haven't discussed, these functions may be useful.

- Whitespace-separated text - `read_table()`
- Tab-separated (TSV) text - `read_tsv()`
- Fixed-width text - `read_fwf()`

Other formats:

To read data that arrives in some other format we haven't discussed, these functions may be useful.

- Whitespace-separated text - `read_table()`
- Tab-separated (TSV) text - `read_tsv()`
- Fixed-width text - `read_fwf()`
- Excel spreadsheet - `read_excel()`

Other formats:

To read data that arrives in some other format we haven't discussed, these functions may be useful.

- Whitespace-separated text - `read_table()`
- Tab-separated (TSV) text - `read_tsv()`
- Fixed-width text - `read_fwf()`
- Excel spreadsheet - `read_excel()`
- SAS archive - `read_sas()`

Other formats:

To read data that arrives in some other format we haven't discussed, these functions may be useful.

- Whitespace-separated text - `read_table()`
- Tab-separated (TSV) text - `read_tsv()`
- Fixed-width text - `read_fwf()`
- Excel spreadsheet - `read_excel()`
- SAS archive - `read_sas()`
- SPSS archive - `read_spss()`

Other formats:

To read data that arrives in some other format we haven't discussed, these functions may be useful.

- Whitespace-separated text - `read_table()`
- Tab-separated (TSV) text - `read_tsv()`
- Fixed-width text - `read_fwf()`
- Excel spreadsheet - `read_excel()`
- SAS archive - `read_sas()`
- SPSS archive - `read_spss()`
- Or just autogenerate code!

Variable names

- Several considerations are often in tension when choosing names for variables

Variable names

- Several considerations are often in tension when choosing names for variables
- Is the name descriptive enough for others (or your future self) to infer its meaning?

Variable names

- Several considerations are often in tension when choosing names for variables
- Is the name descriptive enough for others (or your future self) to infer its meaning?
- Is the name concise enough for easy display?

Variable names

- Several considerations are often in tension when choosing names for variables
- Is the name descriptive enough for others (or your future self) to infer its meaning?
- Is the name concise enough for easy display?
- Is the name consistent with other references to the variable, e.g., in the survey questionnaire or other code?

Variable names

- Several considerations are often in tension when choosing names for variables
- Is the name descriptive enough for others (or your future self) to infer its meaning?
- Is the name concise enough for easy display?
- Is the name consistent with other references to the variable, e.g., in the survey questionnaire or other code?
- Once you've decided on a naming scheme, `dplyr`'s `rename()` function is a good way to rename variables from however they appear in the raw data.

Variable names

```
library(dplyr)
##
## Attaching package: 'dplyr'
## The following objects are masked from 'package:stats':
##
##     filter, lag
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
data(mtcars)
```

Change the names

```
mtcars <- mtcars %>% rename(miles_per_gallon = mpg,
    cylinders = cyl)
# nb the syntax is new name = old name
```

How does it look now?

```
names(mtcars)
##  [1] "miles_per_gallon" "cylinders"        "disp"
##  [4] "hp"               "drat"             "wt"
##  [7] "qsec"             "vs"               "am"
## [10] "gear"             "carb"
```

Data types

- There are four types of data typically encountered in tabular data. Each column in a data.frame has a *class*: `character` for text; `numeric` and its subclass `integer` for numbers; and `logical` for booleans.

Data types

- There are four types of data typically encountered in tabular data. Each column in a data.frame has a *class*: `character` for text; `numeric` and its subclass `integer` for numbers; and `logical` for booleans.
- Managing data types carefully will prevent mistakes. It's possible to store numeric data in a character column, as in `c("1", "2")`, or logical values as characters: `c("TRUE", "FALSE")`. Cleaning data often requires conversion from one type to another. Use the functions `as.numeric()`, `as.logical()`, `as.character`, and `type.convert()`.

Data types

The `factor` class and its subclass `ordered` represent categorical and ordinal data, respectively. It's easy to mistake factor values for strings, but factors are more like integers with labels attached.

```
color <- c("red", "blue", "green")
color_factor <- as.factor(color)
color_factor
## [1] red   blue  green
## Levels: blue green red

# note the lexical sort of levels
levels(color_factor)
## [1] "blue"  "green" "red"

as.integer(color_factor)
## [1] 3 1 2
```

Recoding

When integers in raw data represent levels of a discrete variable (e.g., education in `{1, 2, ..., 5}`), descriptive recoding will help improve readability and avoid human error.

A few examples of this opportunity appear in `dummy_main.dta`.

Recoding

```
library(haven)
survey = read_dta("dummy_main.dta")
glimpse(survey)
## Observations: 483
## Variables: 21
## $ starttime       <dttm> 2016-10-04 17:16:05, 2016-10-07 13:36:22, 2016...
## $ endtime         <dttm> 2016-10-04 20:14:11, 2016-10-07 16:31:31, 2016...
## $ submissiondate <dttm> 2016-10-04 23:32:59, 2016-10-07 19:50:18, 2016...
## $ deviceid        <dbl> 1.351971e+15, 1.351971e+15, 1.351971e+15, 1.351...
## $ subscriberid    <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ simid           <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ key             <chr> "uuid:mpktivgi-lupk-jsjg-cnaq-izqwidogcfzq", "u...
## $ parent_key      <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ surveydate      <date> 2016-10-04, 2016-10-07, 2016-10-07, 2016-09-30...
## $ surveydate2     <date> 2016-10-04, 2016-10-07, 2016-10-07, 2016-09-30...
## $ devicephonenum <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,...
## $ location_code   <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,...
## $ id              <dbl> 1073, 1014, 1010, 1109, 1067, 1104, 1106, 1068,...
## $ surveyor_id     <dbl> 104, 101, 101, 106, 104, 105, 105, 104, 105, 10...
## $ treatment       <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...
## $ gender          <chr> "Female", "Female", "Male", "Male", "Male", "Ma...
## $ name            <chr> "Brenda Greene", "Andrea Carter", "Jonathan Wat...
## $ age_c           <dbl> 4, 5, 14, 10, 12, 11, 8, 14, 14, 8, 12, 10, 7, ...
## $ age_a           <dbl> 39, 39, 25, 41, 43, 39, 32, 33, 25, 24, 32, 25,...
## $ cat_1           <dbl> 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1,...
## $ cat_2           <dbl> 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0,...
```

Recoding

What does the `location_code` variable mean?

```
table(survey$location_code)
##
##   1   2   3   4
## 120 116 120 126
```

Recoding

Suppose we knew that `1`, `2`, `3`, and `4` corresponded with locations `North`, `East`, `South`, and `West`. How could we recode the numbers descriptively? One good way is with the `recode()` function in `dplyr`.

```
survey$location_char <- recode(survey$location_code,
    `1` = "North", `2` = "East", `3` = "South", `4` = "West")
```

Recoding

Did our recoding have the intended effect? Let's use a two-way frequency table to check.

```
with(survey, table(location_code, location_char))
##                location_char
## location_code East North South West
##             1    0   120     0    0
##             2  116     0     0    0
##             3    0     0   120    0
##             4    0     0     0  126
```

Recoding

We could also recode `location_code` as a factor. There are many pitfalls to avoid when working with factors, but an example of this approach is:

```
location_labels = c("North", "East", "South", "West")
survey$location_factor = factor(survey$location_code,
    levels = 1:4, labels = location_labels)
```

Recoding

Check the result:

```
levels(survey$location_factor)
## [1] "North" "East"  "South" "West"
table(survey$location_code, survey$location_factor,
    useNA = "ifany")
##
##        North East South West <NA>
##   1      120    0     0    0    0
##   2        0  116     0    0    0
##   3        0    0   120    0    0
##   4        0    0     0  126    0
##   <NA>     0    0     0    0    1
```

Recoding

See for yourself what happens when you change the order of `levels` and `labels`, or omit observed levels.

To convert a factor to character, use `as.character()`.

Section 3

Data Handling

Dplyr basics

- Pick observations by their values `filter()`

Dplyr basics

- Pick observations by their values `filter()`
- Pick variables by their names `select()`

Dplyr basics

- Pick observations by their values `filter()`
- Pick variables by their names `select()`
- Create new variables with functions of existing variables `mutate()`

Dplyr basics

- Pick observations by their values `filter()`
- Pick variables by their names `select()`
- Create new variables with functions of existing variables `mutate()`
- Collapse many values down to a single summary `summarise()`

Key Grammar

All verbs work similarly:

- The first argument is a data frame.

Key Grammar

All verbs work similarly:

- The first argument is a data frame.
- The subsequent arguments describe what to do with the data frame, using the variable names (without quotes).

Key Grammar

All verbs work similarly:

- The first argument is a data frame.
- The subsequent arguments describe what to do with the data frame, using the variable names (without quotes).
- The result is a new data frame.

First, let's load the dataset

```
library(tidyverse)
library(nycflights13)
head(flights)
## # A tibble: 6 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
## # ... with 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 <dttm>
```

Filter rows with filter()

- dplyr never modifies the original dataset

```
jan1 <- filter(flights, month == 1, day == 1)
```

Small Exercise

- What if we want flights in December 2013?

Small Exercise

- What if we want flights in December 2013?
- dec2013 <- filter(flights, year == 2013, month == 12)

Useful notation

```
filter(flights, month %in% c(11, 12))
```

- x %in% y

Useful notation

```
filter(flights, month %in% c(11, 12))
```

- `x %in% y`
- This will select every row where x is one of the values in y

  What does the previous expression mean?

Select columns with `select()` (1/3)

```
select(flights, year, month, day)
## # A tibble: 336,776 x 3
##     year month   day
##    <int> <int> <int>
##  1  2013     1     1
##  2  2013     1     1
##  3  2013     1     1
##  4  2013     1     1
##  5  2013     1     1
##  6  2013     1     1
##  7  2013     1     1
##  8  2013     1     1
##  9  2013     1     1
## 10  2013     1     1
## # ... with 336,766 more rows
```

Select columns with `select()` (2/3)

```
select(flights, year:day)
## # A tibble: 336,776 x 3
##     year month   day
##    <int> <int> <int>
##  1  2013     1     1
##  2  2013     1     1
##  3  2013     1     1
##  4  2013     1     1
##  5  2013     1     1
##  6  2013     1     1
##  7  2013     1     1
##  8  2013     1     1
##  9  2013     1     1
## 10  2013     1     1
## # ... with 336,766 more rows
```

Select columns with `select()` (3/3)

```
select(flights, -(year:day))
```

What does this give you?

Add new variables with `mutate()`

```
flights_sml <- select(flights, year:day, ends_with("delay"),
    distance, air_time)
```

- `mutate()` always adds new columns at the end of your dataset

Add new variables with `mutate()`

```
flights_sml <- select(flights, year:day, ends_with("delay"),
    distance, air_time)
```

- `mutate()` always adds new columns at the end of your dataset
- Making the dataset smaller to then use this function. What is the code doing?

Using `mutate()`

```
example_1 <- mutate(flights_sml, gain = dep_delay -
    arr_delay, speed = distance/air_time * 60)

example_1[1:5, 7:9]
## # A tibble: 5 x 3
##   air_time  gain speed
##      <dbl> <dbl> <dbl>
## 1      227    -9  370.
## 2      227   -16  374.
## 3      160   -31  408.
## 4      183    17  517.
## 5      116    19  394.
```

Using transmutate()

```
example_2 <- transmute(flights, gain = dep_delay -
    arr_delay, hours = air_time/60, gain_per_hour = gain/hours)

head(example_2)
## # A tibble: 6 x 3
##    gain hours gain_per_hour
##   <dbl> <dbl>         <dbl>
## 1    -9  3.78         -2.38
## 2   -16  3.78         -4.23
## 3   -31  2.67        -11.6
## 4    17  3.05          5.57
## 5    19  1.93          9.83
## 6   -16  2.5           -6.4
```

Grouping and summrizing with `group_by()` and `summarize()`

```
delays <- flights %>% group_by(dest) %>% summarise(count = n(),
    dist = mean(distance, na.rm = TRUE), delay = mean(arr_delay,
        na.rm = TRUE))

delays[1:2, ]
## # A tibble: 2 x 4
##   dest  count  dist delay
##   <chr> <int> <dbl> <dbl>
## 1 ABQ     254  1826  4.38
## 2 ACK     265   199  4.85
```

Grouping and summarizing with `group_by()` and `summarize()`

```
delays <- flights %>% group_by(dest) %>% summarize(count = n(),
    dist = mean(distance, na.rm = TRUE), delay = mean(arr_delay,
        na.rm = TRUE))

delays[1:2, ]
```

- What does `group_by()` do?

Grouping and summarizing with `group_by()` and `summarize()`

```
delays <- flights %>% group_by(dest) %>% summarize(count = n(),
    dist = mean(distance, na.rm = TRUE), delay = mean(arr_delay,
        na.rm = TRUE))

delays[1:2, ]
```

- What does `group_by()` do?
- Remember the %>% operator?

Grouping and summarizing with `group_by()` and `summarize()`

```
delays <- flights %>% group_by(dest) %>% summarize(count = n(),
    dist = mean(distance, na.rm = TRUE), delay = mean(arr_delay,
        na.rm = TRUE))

delays[1:2, ]
```

- What does `group_by()` do?
- Remember the %>% operator?
- What about `summarize()`? The function `mean()` inside it?

Grouping and summarizing with `group_by()` and `summarize()`

```
delays <- flights %>% group_by(dest) %>% summarize(count = n(),
    dist = mean(distance, na.rm = TRUE), delay = mean(arr_delay,
        na.rm = TRUE))

delays[1:2, ]
```

- What does `group_by()` do?
- Remember the %>% operator?
- What about `summarize()`? The function `mean()` inside it?
- Let's talk about `na.rm`

NA Values

- `is.na()` returns true or false. Can use this to subset.

NA Values

- `is.na()` returns true or false. Can use this to subset.
- Try it on the `flights` dataset. What do you get?

NA Values

- `is.na()` returns true or false. Can use this to subset.
- Try it on the `flights` dataset. What do you get?
- What about the function `complete_cases()`? Try it on the dataset, too!