Data Wrangling Strategies with R

Clay Ford

Fall 2021

Agenda

Cover the following topics with examples and exercises:

- Understanding lists
- Working with dates
- Working with character data
- Binding data frames
- Merging or Joining data frames
- Reshaping data frames

Data structures in R: vector

Vector: 1D object of same data type (eg, all numeric, or all character). Like a column of data in a spreadsheet

```
x <- c(1, 4, 7, 11)
x
```

```
## [1]  1  4  7 11
```

Data structures in R: data frame

data frame: 2D object consisting of vectors of the same length but of potentially different data types

```r
x <- c(1, 3, 5)
g <- c("M", "M", "F")
df <- data.frame(x, g)
df
```

```
##   x g
## 1 1 M
## 2 3 M
## 3 5 F
```

Notice `x` is numeric while `g` is character.

Data structures in R: lists

list: the most general data structure. It can contain vectors, data.frames, and other lists. When data wrangling, we sometimes need to work with lists at an intermediate step.

```
lst <- list(x = 5, df = data.frame(x, g))
lst
```

```
## $x
## [1] 5
##
## $df
##   x g
## 1 1 M
## 2 3 M
## 3 5 F
```

Applying functions to elements in a list

The `lapply` function allows us to *apply* functions to elements of a list. Example: Find the mean of each list element

```
(lst <- list(y1 = 1:4, y2 = 6:10))
```

```
## $y1
## [1] 1 2 3 4
##
## $y2
## [1]  6  7  8  9 10
```

```
lapply(lst, mean)
```

```
## $y1
## [1] 2.5
##
## $y2
## [1] 8
```

Dates and Date-times in R

- When a date (eg, April 5, 1982) is formatted as a “date” in R, it becomes the number of days after (or before) January 1, 1970.
- When a date-time (eg, April 5, 1982 1:15 PM) is formatted as a “date-time”, it becomes the number of seconds after (or before) January 1, 1970.
- This simplifies the calculation of time spans.
- The `lubridate` pacakge helps us parse dates and date-times as well as perform calculations and conversions of such values.

Parsing dates

- `lubridate` provides a series of functions for parsing dates that are a permutation of the letters “m”, “d” and “y” to represent the ordering of month, day and year.
- `lubridate` provides functions for every permutation of “m”, “d”, “y”.

```
library(lubridate)
d <- "April 5, 1982"
d <- mdy(d)
d
```

```
## [1] "1982-04-05"
```

Printed values vs. stored values

- A date that is parsed with `lubridate` will print to the console and appear in data frames as if it's character data.
- Use `as.numeric` to see the stored value.

```
d
```

```
## [1] "1982-04-05"
```

```
as.numeric(d) # days since 1/1/1970
```

```
## [1] 4477
```

Parsing date-times

To parse date-times, append either `_h`, `_hm`, or `_hms` to the “mdy” function.

```
d <- "April 5, 1982 1:15 PM"
d <- mdy_hm(d)
d
```

```
## [1] "1982-04-05 13:15:00 UTC"
```

```
as.numeric(d) # seconds since 1/1/1970
```

```
## [1] 386860500
```

Parsing times

`lubridate` also allows us to parse hours, minutes and seconds using `hm`, `ms` and `hms`

```
t <- c("1:23","2:34")
hm(t)
```

```
## [1] "1H 23M 0S" "2H 34M 0S"
```

```
ms(t)
```

```
## [1] "1M 23S" "2M 34S"
```

The output is nicely formatted, but these are stored as seconds

Extracting date components

`lubridate` provides functions such as `month`, `day`, `wday`, `yday`, etc. to extract date components

```r
d <- mdy("5/5/01")
month(d, label = TRUE) # month
```

```
## [1] May
## 12 Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug
```

```r
wday(d, label = TRUE) # week day
```

```
## [1] Sat
## Levels: Sun < Mon < Tue < Wed < Thu < Fri < Sat
```

Character strings

Our data often include character strings such as names, locations, descriptions, categories or unwanted “junk”. Examples of manipulating character strings include. . .

- Converting text to UPPERCASE or lowercase
- Extract parts of a string (Eg, extract “23” from “`<b>23</b>`”)
- Padding strings with zeroes, so 9, 10, 11 become 009, 010, 011
- Identify patterns of text for purpose of extracting, replacing or subsetting data

We will use the `stringr` package to work with character strings.

Character data in R

- Character data have quotes when printed to console
- But data with quotes does not mean it's character!
- use `is.character()` to find out.
- Character data need to be surrounded with quotes (either single or double) when used in `R` code

```
(x <- c("a","b","c","12"))
```

```
## [1] "a"  "b"  "c"  "12"
```

```
is.character(x)
```

```
## [1] TRUE
```

Character versus factor

- Sometimes data that appear to be character are actually stored as a *factor*
- factors are character data that are stored as integers but have character labels
- factors are good for using character data in statistical modeling (eg, ANOVA, regression, etc)
- If your character data is stored as a factor, R automatically handles conversion to dummy matrices necessary for statistical modeling routines
- factors do not have quotes when printed to console

Factor data in R

```
(y <- factor(c("a","b","c","c")))
```

```
## [1] a b c c
## Levels: a b c
```

```
is.character(y)
```

```
## [1] FALSE
```

When to convert factors to character

- If you plan to clean or manipulate character data, make sure it's character, not factor.
- Change factor to character with `as.character` function

```
(y <- factor(c("a","b","c","c")))
```

```
## [1] a b c c
## Levels: a b c
```

```
(y <- as.character(y))
```

```
## [1] "a" "b" "c" "c"
```

Convert case of string

- `str_to_upper`, `str_to_lower`, and `str_to_title` do what you expect

```r
library(stringr)
str_to_upper("day one")
```

```
## [1] "DAY ONE"
```

```r
str_to_lower("DAY ONE")
```

```
## [1] "day one"
```

```r
str_to_title("day one")
```

```
## [1] "Day One"
```

find-and-remove within strings

- `str_remove` finds first occurrence of specified pattern and removes it

```r
# find first - and replace with nothing
str_remove("434-555-1212", "-")
```

```
## [1] "434555-1212"
```

- `str_remove_all` finds all occurrences of specified pattern and removes it

```r
# find all - and replace with nothing
str_remove_all("434-555-1212", "-")
```

```
## [1] "4345551212"
```

find-and-replace within strings

- `str_replace` finds first occurrence of specified pattern and replaces with specified text

```
str_replace("434-555-1212", "-", ".")
```

```
## [1] "434.555-1212"
```

- `str_replace_all` finds all occurrences of specified pattern and replaces with specified text

```
str_replace_all("434-555-1212", "-", ".")
```

```
## [1] "434.555.1212"
```

Pad a string with characters

- `str_pad` will pad a string with characters. This is useful for zip codes or ID numbers.
- Specify the width, the side of the padding, and what to pad with.

```
(zips <- c(22904, 06443, 01331))
```

```
## [1] 22904  6443  1331
```

```
str_pad(zips, width = 5, side = "left", pad = "0")
```

```
## [1] "22904" "06443" "01331"
```

Regular Expressions

- Regular expressions are a language for describing text patterns (eg: email addresses, social security numbers, html tags)
- A regular expression is usually formed with some combination of *literal characters*, *character classes* and *modifiers*
  - literal character example: `state` (looking for `"state"`)
  - character class example: `[0-9]` (any number 0 - 9)
  - modifier example: + (1 or more of whatever it follows)
- Regular expression example: `state[0-9]+` finds patterns such as `state1`, `state12`, `state99` but not `state`
- We will cover just the basics today as they work in R

Character classes

- `[0-9]`, `[a-z]`, `[A-Z]`
- Define your own: `[0-3a-g]`, `[AEIOUaeiou]`
- Predefined character classes
  - `[:alpha:]` all letters
  - `[:digit:]` numbers 0 - 9
  - `[:alnum:]` Alphanumeric characters (alpha and digit)
  - `[:blank:]` Blank characters: space and tab
  - `[:lower:]` lowercase letters
  - `[:upper:]` UPPERCASE letters
  - `[:punct:]` Punctuation characters
  - `[:print:]` Printable characters: [:alnum:], [:punct:] and space
  - `[:space:]` Space characters: tab, newline, vertical tab, form feed, carriage return, space

Modifiers

- `^` start of string; or negation inside character class
- `$` end of string
- `.` any character except new line
- `*` 0 or more
- `+` 1 or more
- `?` 0 or 1
- `|` or (alternative patterns)
- `{}` quantifier brackets: exactly `{n}`; at least `{n,}`; between `{n,m}`
- `()` group patterns together
- `\` escape character (needs to be escaped itself in R! `\\`)
- `[]` character class brackets

Note: precede these with a double backslash if you want to treat them as literal characters.

Basic regular expression examples

- Remove one or more letters followed by `.` and space at beginning of string

```
names <- c("Dr. Claibourn","Mr. Ford","Ms. Draber")
str_remove(names, pattern = "^[:alpha:]+\\. ")
```

```
## [1] "Claibourn" "Ford"      "Draber"
```

- Replace “Group”, “grp”, and “group” with “G”

```
group <- c("Group 1", "grp 1", "group 3")
str_replace(group, "(Group |grp |group )", "G")
```

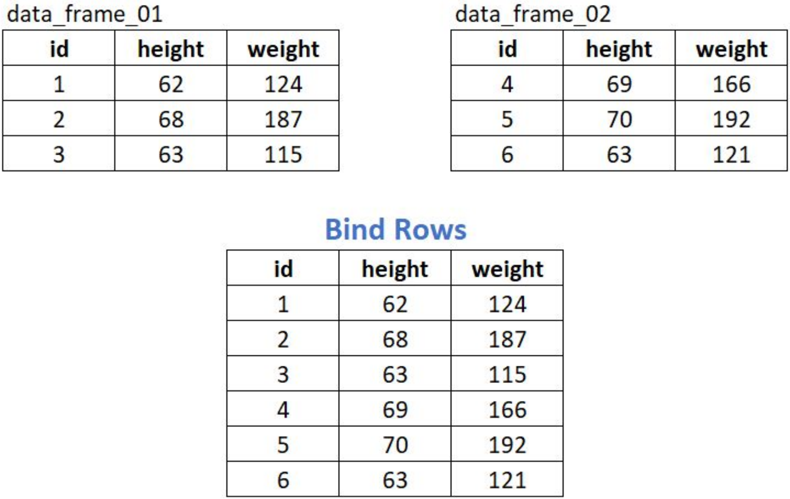
```
## [1] "G1" "G1" "G3"
```

Binding data frames

- Row binding: stacking data frames on top of one another
- Column binding: setting data frames next to each other
- Row binding is more common; often used when reading in multiple files of matching structure that need to be combined into one data frame
- Example: importing 10 Excel worksheets for years 2000 - 2009, and then combining into one data frame
- `dplyr` functions: `bind_rows` and `bind_cols`

Row binding

```
bind_rows(data_frame_01, data_frame_02)
```



data_frame_01

id	height	weight
1	62	124
2	68	187
3	63	115

data_frame_02

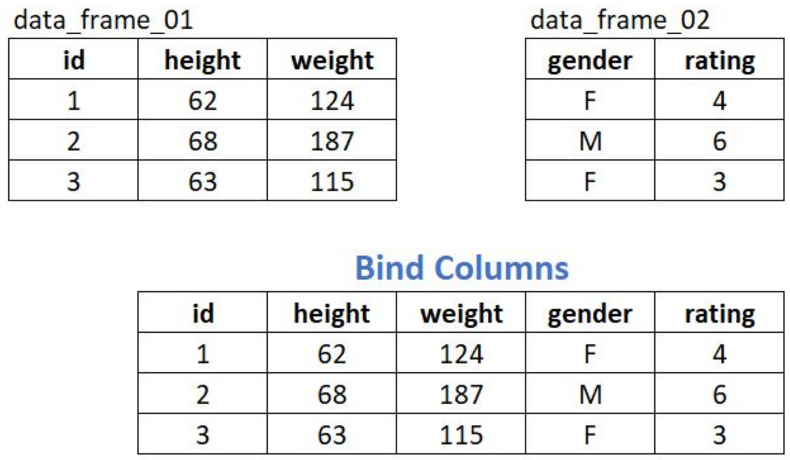
id	height	weight
4	69	166
5	70	192
6	63	121

Bind Rows

id	height	weight
1	62	124
2	68	187
3	63	115
4	69	166
5	70	192
6	63	121

Column binding

```
bind_cols(data_frame_01, data_frame_02)
```
[1]



data_frame_01

id	height	weight
1	62	124
2	68	187
3	63	115

data_frame_02

gender	rating
F	4
M	6
F	3

Bind Columns

id	height	weight	gender	rating
1	62	124	F	4
2	68	187	M	6
3	63	115	F	3

[1] data frames must have the same number of rows.

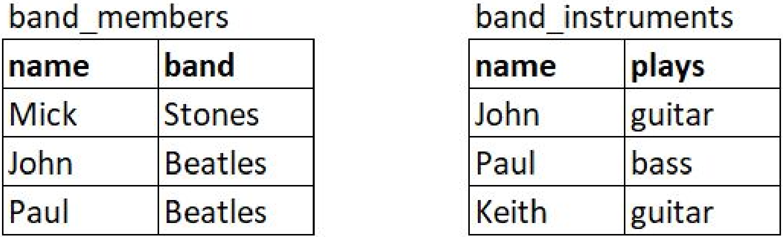
Merging or Joining data frames

- Two types of merges, or joins:
  - **Mutating join**: Join data frames based on a common column (or columns), called “keys”
  - **Filtering join**: keep rows in one data frame based on having (or not having) membership in another data frame
- Frequently used to combine two different sources of data.
- Example: merge subject demographic data with subject lab data
- `dplyr` functions: `inner_join`, `left_join`, `right_join`, `full_join`, `semi_join`, `anti_join`

Inner Join

Retain only those rows with matching keys in both data sets.

```
inner_join(band_members,band_instruments,by="name")
```

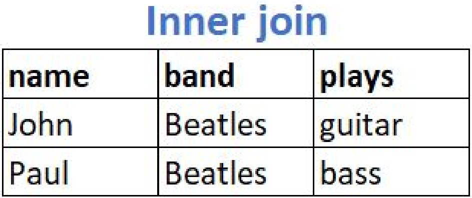


band_members

name	band
Mick	Stones
John	Beatles
Paul	Beatles

band_instruments

name	plays
John	guitar
Paul	bass
Keith	guitar



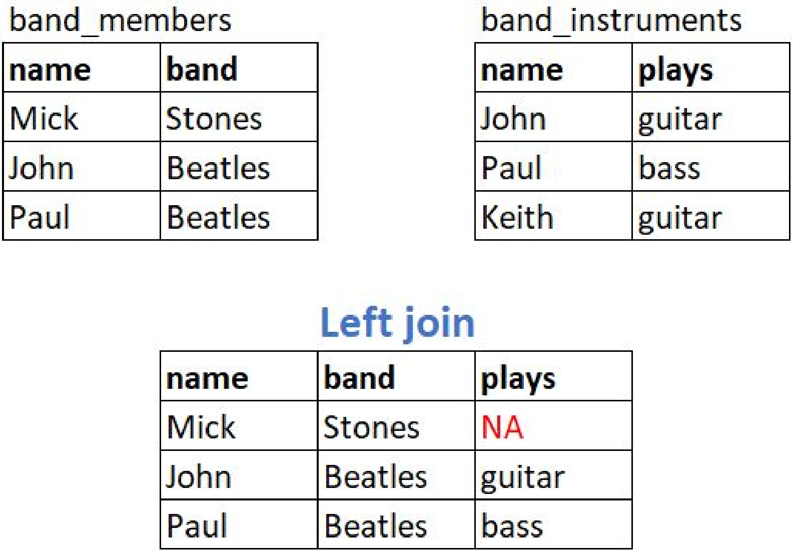
Inner join

name	band	plays
John	Beatles	guitar
Paul	Beatles	bass

Left Join

Retain everything in the left data set, merge matches from right.

```
left_join(band_members,band_instruments,by="name")
```



band_members

name	band
Mick	Stones
John	Beatles
Paul	Beatles

band_instruments

name	plays
John	guitar
Paul	bass
Keith	guitar

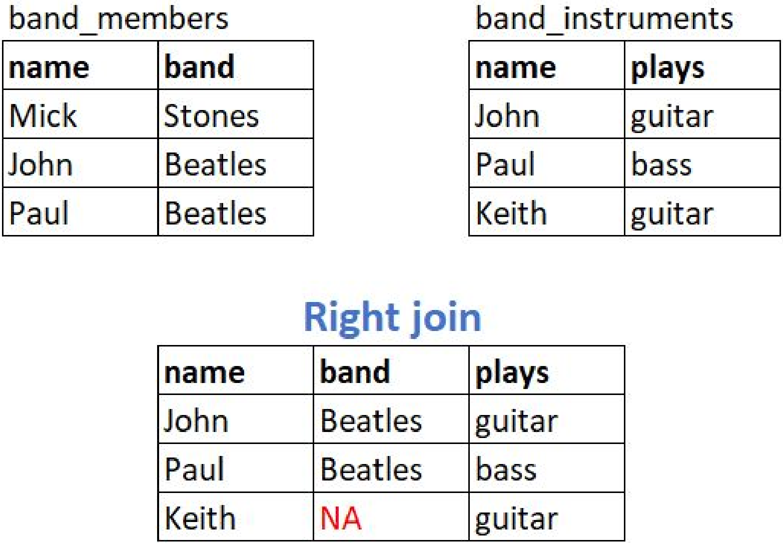
Left join

name	band	plays
Mick	Stones	NA
John	Beatles	guitar
Paul	Beatles	bass

Right Join

Retain everything in the right data set, merge from left.

```
right_join(band_members,band_instruments,by="name")
```



band_members

name	band
Mick	Stones
John	Beatles
Paul	Beatles

band_instruments

name	plays
John	guitar
Paul	bass
Keith	guitar

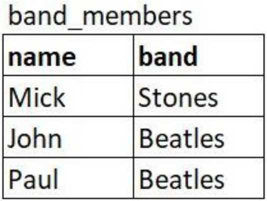
Right join

name	band	plays
John	Beatles	guitar
Paul	Beatles	bass
Keith	NA	guitar

Full Join

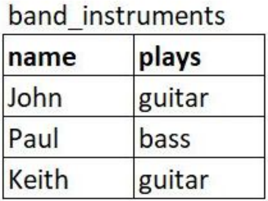
Retain all rows in both data sets.

```
full_join(band_members,band_instruments,by="name")
```



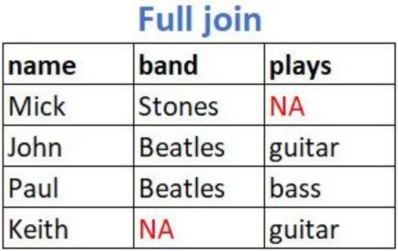
band_members

name	band
Mick	Stones
John	Beatles
Paul	Beatles



band_instruments

name	plays
John	guitar
Paul	bass
Keith	guitar



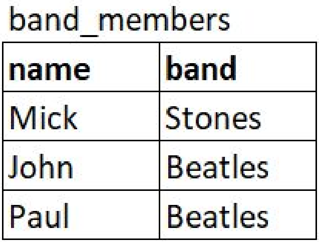
Full join

name	band	plays
Mick	Stones	NA
John	Beatles	guitar
Paul	Beatles	bass
Keith	NA	guitar

Semi Join

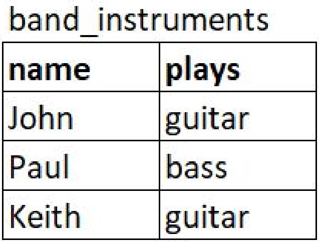
Retain all rows from left that have matching values in right.

```
semi_join(band_members,band_instruments,by="name")
```



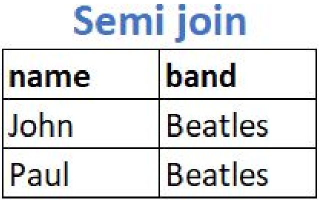
band_members

name	**band**
Mick	Stones
John	Beatles
Paul	Beatles



band_instruments

name	**plays**
John	guitar
Paul	bass
Keith	guitar



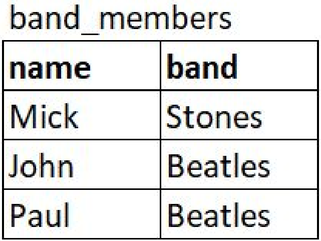
Semi join

name	**band**
John	Beatles
Paul	Beatles

Anti Join

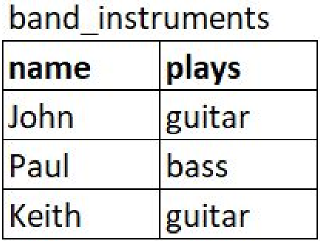
Retain all rows from left that do NOT have matching values in right.

```
anti_join(band_members,band_instruments,by="name")
```



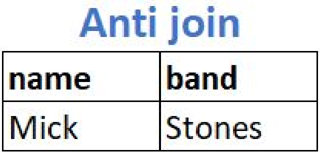
band_members

name	band
Mick	Stones
John	Beatles
Paul	Beatles



band_instruments

name	plays
John	guitar
Paul	bass
Keith	guitar



Anti join

name	band
Mick	Stones

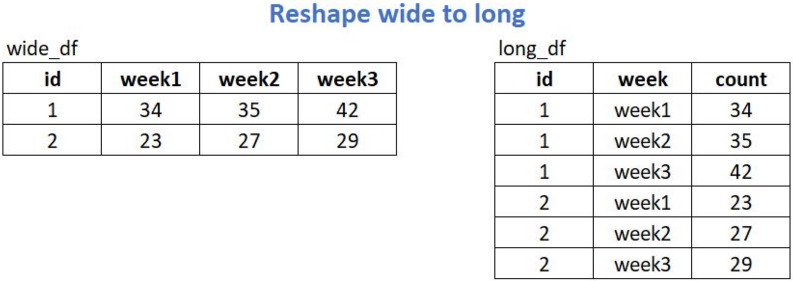
Reshaping data frames

- Taking column names and making them values in a single column, and vice versa; similiar to transposing data in Excel
- Often expressed as reshaping “wide to long”, or “long to wide”
- Reshaping wide to long is very common in R; often needed to accommodate modeling and plotting functions
- `tidyr` functions: `pivot_longer` (wide to long) and `pivot_wider` (long to wide)

Reshaping wide to long

Reshape columns week1 - week3 into two columns, one for the column headers (`names_to`) and the other for the values in the columns (`values_to`).

```
pivot_longer(wide_df, week1:week3,
             names_to = "week, values_to = "count)
```



Reshape wide to long

wide_df

id	week1	week2	week3
1	34	35	42
2	23	27	29

long_df

id	week	count
1	week1	34
1	week2	35
1	week3	42
2	week1	23
2	week2	27
2	week3	29

References

R for Data Science: http://r4ds.had.co.nz/

Free online edition of the O'Reilly book *R for Data Science*, by Garrett Grolemund and Hadley Wickham.

The UVa Library also has a physical copy.

See also

This workshop was previously offered as two workshops, where each went into more detail. Here are links to the materials.

Part 1: bind, merge, reshape
http://bit.ly/dwr_01

Part 2: dates and strings
http://bit.ly/dwr_02

Thanks for coming

- For statistical consulting: statlab@virginia.edu
- Sign up for more workshops or see past workshops: http://data.library.virginia.edu/training/
- Register for the Research Data Services newsletter to be notified of new workshops: http://data.library.virginia.edu/newsletters/