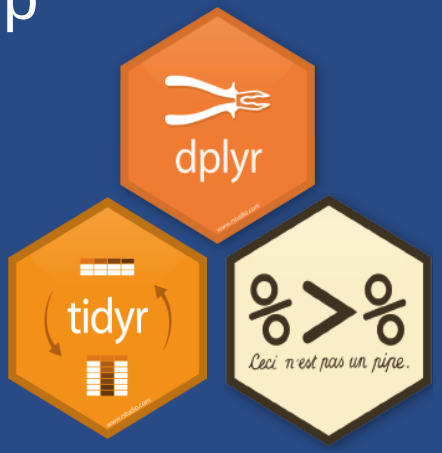
tidyr, dplyr, and the pipe %>%

University of Guelph R Users Group

Joey Burant

October 2018



What is tidy data?

"Happy families are all alike; every unhappy family is unhappy in its own way." -- Tolstoy

- Data cleaning/wrangling/munging/tidying is the most time-intensive component of data analysis

- In tidy data:
  - each variable forms a column
  - each observation forms a row
  - each type of observational unit forms a table
  - data frames are rectangles

Rect
Data Wrangling

tidyr

- Useful functions in tidyr:
  - gather() -- take multiple columns and collect them into a value-key pair (format = wide → long)
    - functionally-equivalent to reshape2::melt()
  - spread() -- take two columns and spread into multiple columns (format = long → wide)
    - functionally-equivalent to reshape2::cast()
- Other functions: separate(), unite()

dplyr

- Useful functions in dplyr:
  - mutate() -- create a new variable (column) that is a function of existing variables
  - select() -- choose columns based on their names (or locations)
    - useful for subsetting to remove extraneous columns (although it's usually better to keep them all!)
  - filter() -- choose rows based on their value
    - useful for subsetting to include only a selection of records (e.g., if you have multiple individuals/groups and only want to look at data for some of them)
- Other functions: summarise()/summarize(), arrange()

What is piping?

- Have you ever needed to perform multiple operations on the same dataset or variable? (Of course you have!)

How do you usually do this?

- Two common, non-piped approaches:
  - Step-by-step

```
data2 <- mutate(data, varZ = varA * varB)
data3 <- filter(data2, varA > varB & varZ < 100)
```

  - Nested

```
data <- filter(mutate(data, varZ = varA * varB), varA > varB & varZ < 100))
```

What is piping?

- The pipe %>% comes from magrittr, which loads automatically when you call tidyverse in R

- A pipe is a powerful way of clearly expressing a sequences of operations or steps:
  - Using object X, first do A, then do B, then…

- Helps write more concise and readable code

Ceci n'est pas un pipe.

Little bunny Foo Foo
Went **hopping** through the forest
Scooping up the field mice
And **bopping** them on the head

```
foo_foo <- bop(scoop(hop(little_bunny(), through = forest)), up = field_mice), on = head)
```

```
foo_foo <- little_bunny()

foo_foo1 <- hop(foo_foo, through = forest)

foo_foo2 <- scoop(foo_foo1, up = field_mice)

foo_foo3 <- bop(foo_foo2, on = head)
```

```
foo_foo %>%
  hop(through = forest) %>%
  scoop(up = field_mice) %>%
  bop(on = head)
```

Resources

- tidyr
  - https://tidyr.tidyverse.org
  - https://cran.r-project.org/web/packages/tidyr/tidyr.pdf
  - http://vita.had.co.nz/papers/tidy-data.html
- dplyr
  - https://dplyr.tidyverse.org
  - https://cran.r-project.org/web/packages/dplyr/dplyr.pdf
- piping (%>%)
  - http://r4ds.had.co.nz/pipes.html
  - https://www.datacamp.com/community/tutorials/pipe-r-tutorial

The tidyverse awaits...

https://www.tidyverse.org

When not to use the pipe %>%

You might consider a different approach if:

- Your pipes are longer than (say) ten steps. In that case, create intermediate objects with meaningful names. That will make debugging easier, because you can more easily check the intermediate results, and it makes it easier to understand your code, because the variable names can help communicate intent.
- You have multiple inputs or outputs. If there isn't one primary object being transformed, but two or more objects being combined together, don't use the pipe.
- You are starting to think about a directed graph with a complex dependency structure. Pipes are fundamentally linear and expressing complex relationships with them will typically yield confusing code.