Introduction to dplyr

Jenny Bryan, modified by Roger H. French

29 September, 2018

Contents

5.2.2.1 Intro

`dplyr` is a new package for data manipulation.

- It is part of the tidyverse,
  - and is loaded with the tidyverse metapackage
- It is built to be fast, highly expressive, and open-minded
  - about how your data is stored.
- It is developed by Hadley Wickham and Romain Francois.

`dplyr`'s roots are in an earlier, still-very-useful package

- called `plyr`,
- which implements the "split-apply-combine" strategy for data analysis.

Where `plyr` covers a diverse set of inputs and outputs

- (e.g., arrays, data.frames, lists),
- `dplyr` has a laser-like focus on data.frames and related structures.

Have no idea what I'm talking about?

- Not sure if you care?
- If you use these base R functions:
  - `subset()`, `apply()`, `[sl]apply()`, `tapply()`,
  - `aggregate()`, `split()`, `do.call()`,
- then you should keep reading.

5.2.2.1.1 Load `dplyr`

```
## install if you do not already have

## from CRAN:
## install.packages('dplyr')

## from GitHub using devtools (which you also might need to install!):
```

```
## devtools::install_github("hadley/lazyeval")
## devtools::install_github("hadley/dplyr")
## suppressPackageStartupMessages(library(dplyr))
```

```
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
```

5.2.2.1.2 Load the Gapminder data

An excerpt of the Gapminder data which we work with alot.

```
gd_url <- "http://www.stat.ubc.ca/~jenny/notOcto/STAT545A/examples/gapminder/data/gapminderDataFiveYear
# gd_url <- "http://tiny.cc/gapminder"
gdf <- read.delim(file = gd_url)
str(gdf)
```

```
## 'data.frame':    1704 obs. of  6 variables:
##  $ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ year     : int  1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
##  $ pop      : num  8425333 9240934 10267083 11537966 13079460 ...
##  $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
##  $ gdpPercap: num  779 821 853 836 740 ...
```

```
head(gdf)
```

```
##       country year      pop continent lifeExp gdpPercap
## 1 Afghanistan 1952  8425333      Asia  28.801  779.4453
## 2 Afghanistan 1957  9240934      Asia  30.332  820.8530
## 3 Afghanistan 1962 10267083      Asia  31.997  853.1007
## 4 Afghanistan 1967 11537966      Asia  34.020  836.1971
## 5 Afghanistan 1972 13079460      Asia  36.088  739.9811
## 6 Afghanistan 1977 14880372      Asia  38.438  786.1134
```

5.2.2.2 Meet `tbl_df`, an upgrade to `data.frame`

```
gtbl <- tbl_df(gdf)
gtbl
```

```
## # A tibble: 1,704 x 6
##    country      year      pop continent lifeExp gdpPercap
##    <fct>       <int>    <dbl> <fct>       <dbl>     <dbl>
##  1 Afghanistan  1952  8425333 Asia         28.8      779.
##  2 Afghanistan  1957  9240934 Asia         30.3      821.
##  3 Afghanistan  1962 10267083 Asia         32.0      853.
##  4 Afghanistan  1967 11537966 Asia         34.0      836.
```

```
##  5 Afghanistan  1972 13079460 Asia         36.1      740.
##  6 Afghanistan  1977 14880372 Asia         38.4      786.
##  7 Afghanistan  1982 12881816 Asia         39.9      978.
##  8 Afghanistan  1987 13867957 Asia         40.8      852.
##  9 Afghanistan  1992 16317921 Asia         41.7      649.
## 10 Afghanistan  1997 22227415 Asia         41.8      635.
## # ... with 1,694 more rows
```

```
glimpse(gtbl)
```

```
## Observations: 1,704
## Variables: 6
## $ country   <fct> Afghanistan, Afghanistan, Afghanistan, Afghanistan, ...
## $ year      <int> 1952, 1957, 1962, 1967, 1972, 1977, 1982, 1987, 1992...
## $ pop       <dbl> 8425333, 9240934, 10267083, 11537966, 13079460, 1488...
## $ continent <fct> Asia, Asia, Asia, Asia, Asia, Asia, Asia, Asia, Asia...
## $ lifeExp   <dbl> 28.801, 30.332, 31.997, 34.020, 36.088, 38.438, 39.8...
## $ gdpPercap <dbl> 779.4453, 820.8530, 853.1007, 836.1971, 739.9811, 78...
```

A `tbl_df` is basically an improved data.frame,

- or a tibble dataframe
- for which `dplyr` provides nice methods for high-level inspection.

Specifically, these methods do something sensible

- for datasets with many observations and/or variables.
- You do **NOT** need to turn your data.frames
  - into `tbl_dfs` to use `plyr`.
- I do so here for demonstration purposes only.

5.2.2.3 Think before you create excerpts of your data ...

If you feel the urge to store a little snippet of your data:

```
(snippet <- subset(gdf, country == "Canada"))
```

```
##     country year      pop continent lifeExp gdpPercap
## 241  Canada 1952 14785584  Americas  68.750  11367.16
## 242  Canada 1957 17010154  Americas  69.960  12489.95
## 243  Canada 1962 18985849  Americas  71.300  13462.49
## 244  Canada 1967 20819767  Americas  72.130  16076.59
## 245  Canada 1972 22284500  Americas  72.880  18970.57
## 246  Canada 1977 23796400  Americas  74.210  22090.88
## 247  Canada 1982 25201900  Americas  75.760  22898.79
## 248  Canada 1987 26549700  Americas  76.860  26626.52
## 249  Canada 1992 28523502  Americas  77.950  26342.88
## 250  Canada 1997 30305843  Americas  78.610  28954.93
## 251  Canada 2002 31902268  Americas  79.770  33328.97
## 252  Canada 2007 33390141  Americas  80.653  36319.24
```

Stop and ask yourself ...

> Do I want to create mini datasets for each level of some factor (or unique combination of several factors) ... in order to compute or graph something?

If YES, **use proper data aggregation techniques**

- or facetting in `ggplot2` plots

- or conditioning in `lattice`

– **don't subset the data**.

Or, more realistic,

- only subset the data as a temporary measure
- while you develop your elegant code
  - for computing on or visualizing these data subsets.

If NO, then maybe you really do need to store a copy of a subset of the data.

- But seriously consider whether you can achieve your goals
- by simply using the `subset =` argument of,
  - e.g., the `lm()` function,
  - to limit computation to your excerpt of choice.
- Lots of functions offer a `subset =` argument!

Copies and excerpts of your data

- clutter your workspace,
- invite mistakes,
- and sow general confusion.
- Avoid whenever possible.

Reality can also lie somewhere in between.

- You will find the workflows presented below
  - can help you accomplish your goals
  - with minimal creation of temporary, intermediate objects.

5.2.2.4 Use `filter()` to subset data row-wise.

`filter()` takes logical expressions

- and returns the rows for which all are `TRUE`.

```r
filter(gtbl, lifeExp < 29)
```

```
## # A tibble: 2 x 6
##   country      year     pop continent lifeExp gdpPercap
##   <fct>       <int>   <dbl> <fct>       <dbl>     <dbl>
## 1 Afghanistan  1952 8425333 Asia         28.8      779.
## 2 Rwanda       1992 7290203 Africa       23.6      737.
```

```r
filter(gtbl, country == "Rwanda")
```

```
## # A tibble: 12 x 6
##    country  year     pop continent lifeExp gdpPercap
##    <fct>   <int>   <dbl> <fct>       <dbl>     <dbl>
##  1 Rwanda   1952 2534927 Africa       40        493.
##  2 Rwanda   1957 2822082 Africa       41.5      540.
##  3 Rwanda   1962 3051242 Africa       43        597.
##  4 Rwanda   1967 3451079 Africa       44.1      511.
##  5 Rwanda   1972 3992121 Africa       44.6      591.
##  6 Rwanda   1977 4657072 Africa       45        670.
##  7 Rwanda   1982 5507565 Africa       46.2      882.
##  8 Rwanda   1987 6349365 Africa       44.0      848.
##  9 Rwanda   1992 7290203 Africa       23.6      737.
## 10 Rwanda   1997 7212583 Africa       36.1      590.
```

```
## 11 Rwanda   2002 7852401 Africa       43.4      786.
## 12 Rwanda   2007 8860588 Africa       46.2      863.
```

```
filter(gtbl, country %in% c("Rwanda", "Afghanistan"))
```

```
## # A tibble: 24 x 6
##    country      year      pop continent lifeExp gdpPercap
##    <fct>       <int>    <dbl> <fct>       <dbl>     <dbl>
##  1 Afghanistan  1952  8425333 Asia         28.8      779.
##  2 Afghanistan  1957  9240934 Asia         30.3      821.
##  3 Afghanistan  1962 10267083 Asia         32.0      853.
##  4 Afghanistan  1967 11537966 Asia         34.0      836.
##  5 Afghanistan  1972 13079460 Asia         36.1      740.
##  6 Afghanistan  1977 14880372 Asia         38.4      786.
##  7 Afghanistan  1982 12881816 Asia         39.9      978.
##  8 Afghanistan  1987 13867957 Asia         40.8      852.
##  9 Afghanistan  1992 16317921 Asia         41.7      649.
## 10 Afghanistan  1997 22227415 Asia         41.8      635.
## # ... with 14 more rows
```

Compare with some base R code

- to accomplish the same things

```
gdf[gdf$lifeExp < 29, ] ## repeat `gdf`, [i, j] indexing is distracting
subset(gdf, country == "Rwanda") ## almost same as filter ... but wait ...
```

5.2.2.5 Meet the new pipe operator

Before we go any further,

- we should exploit the new pipe operator
- that `dplyr` imports from the `magrittr` package.

This changes your data analytical life.

You no longer need to enact multi-operation commands

- by nesting them inside each other,
- like so many Russian nesting dolls.

This new syntax leads to code

- that is much easier to write and to read.

Here's what it looks like: `%>%`.

The RStudio keyboard shortcut:

- Ctrl + Shift + M (Linux/Windows),
- Cmd + Shift + M (Mac),
- according to this tweet.

Let's demo then I'll explain:

```
gdf %>% head
```

```
##       country year      pop continent lifeExp gdpPercap
## 1 Afghanistan 1952  8425333      Asia  28.801  779.4453
## 2 Afghanistan 1957  9240934      Asia  30.332  820.8530
## 3 Afghanistan 1962 10267083      Asia  31.997  853.1007
```

```
## 4 Afghanistan 1967 11537966      Asia  34.020  836.1971
## 5 Afghanistan 1972 13079460      Asia  36.088  739.9811
## 6 Afghanistan 1977 14880372      Asia  38.438  786.1134
```

This is equivalent to `head(gdf)`.

- This pipe operator takes the thing on the left-hand-side
- and **pipes** it into the function call on the right-hand-side -literally, it drops it in as the first argument.

Never fear, you can still specify other arguments to this function!

To see the first 3 rows of Gapminder,

- we could say `head(gdf, 3)`
- or this:

```
gdf %>% head(3)
```

```
##       country year      pop continent lifeExp gdpPercap
## 1 Afghanistan 1952  8425333      Asia  28.801  779.4453
## 2 Afghanistan 1957  9240934      Asia  30.332  820.8530
## 3 Afghanistan 1962 10267083      Asia  31.997  853.1007
```

I've advised you to think "gets"

- whenever you see the assignment operator, `<-`.

Similary, you should think "then"

- whenever you see the pipe operator, `%>%`.

You are probably not impressed yet,

- but the magic will soon happen.

5.2.2.6 Use `select()` to subset the data on variables or columns.

Back to `dplyr` . . .

Use `select()` to subset the data

- on variables or columns.

Here's a conventional call:

```
select(gtbl, year, lifeExp) ## tbl_df prevents TMI from printing
```

```
## # A tibble: 1,704 x 2
##     year lifeExp
##    <int>   <dbl>
##  1  1952    28.8
##  2  1957    30.3
##  3  1962    32.0
##  4  1967    34.0
##  5  1972    36.1
##  6  1977    38.4
##  7  1982    39.9
##  8  1987    40.8
##  9  1992    41.7
## 10  1997    41.8
## # ... with 1,694 more rows
```

And here's similar operation,

- but written with the pipe operator
- and piped through `head`:

```
gtbl %>%
  select(year, lifeExp) %>%
  head(4)
```

```
## # A tibble: 4 x 2
##    year lifeExp
##   <int>   <dbl>
## 1  1952    28.8
## 2  1957    30.3
## 3  1962    32.0
## 4  1967    34.0
```

Think:

- “Take `gtbl`,
- then select the variables year and lifeExp,
- then show the first 4 rows."

5.2.2.7 Revel in the convenience

Here’s the data for Cambodia,

- but only certain variables:

```
gtbl %>%
  filter(country == "Cambodia") %>%
  select(year, lifeExp)
```

```
## # A tibble: 12 x 2
##     year lifeExp
##    <int>   <dbl>
##  1  1952    39.4
##  2  1957    41.4
##  3  1962    43.4
##  4  1967    45.4
##  5  1972    40.3
##  6  1977    31.2
##  7  1982    51.0
##  8  1987    53.9
##  9  1992    55.8
## 10  1997    56.5
## 11  2002    56.8
## 12  2007    59.7
```

and what a typical base R call would look like:

```
gdf[gdf$country == "Cambodia", c("year", "lifeExp")]
```

```
##     year lifeExp
## 217 1952  39.417
## 218 1957  41.366
## 219 1962  43.415
## 220 1967  45.415
## 221 1972  40.317
## 222 1977  31.220
```

```
## 223 1982  50.957
## 224 1987  53.914
## 225 1992  55.803
## 226 1997  56.534
## 227 2002  56.752
## 228 2007  59.723
```

or, possibly?, a nicer look using base R's `subset()` function:

```r
subset(gdf, country == "Cambodia", select = c(year, lifeExp))
```

```
##     year lifeExp
## 217 1952  39.417
## 218 1957  41.366
## 219 1962  43.415
## 220 1967  45.415
## 221 1972  40.317
## 222 1977  31.220
## 223 1982  50.957
## 224 1987  53.914
## 225 1992  55.803
## 226 1997  56.534
## 227 2002  56.752
## 228 2007  59.723
```

5.2.2.8 Pause to reflect

We've barely scratched the surface of `dplyr`

- but I want to point out key principles you may start to appreciate.

`dplyr`'s verbs,

- such as `filter()` and `select()`,
- are what's called pure functions. To quote from Wickham's Advanced R Programming book:

" The functions that are the easiest to understand and reason about

- are pure functions:
  - functions that always map the same input to the same output
  - and have no other impact on the workspace. In other words, pure functions have no side effects:
  - they don't affect the state of the world in any way
  - apart from the value they return."

In fact, these verbs are a special case of pure functions:

- they take the same flavor of object as input and output.
- Namely, a data.frame or one of the other data receptacles `dplyr` supports.
- And finally, the data is **always** the very first argument
  - of the verb functions.

This set of deliberate design choices,

- together with the new pipe operator,
- produces a highly effective,
- low friction domain-specific language
- for data analysis.

5.2.2.9 Links

Jenny Bryan Stat 545

`dplyr` official stuff

- package home on CRAN
  - note there are several vignettes, with the introduction being the most relevant right now
  - the one on window functions will also be interesting to you now
- development home on GitHub
- tutorial HW delivered (note this links to a DropBox folder) at useR! 2014 conference

Blog post Hands-on dplyr tutorial for faster data manipulation in R by Data School, that includes a link to an R Markdown document and links to videos

Cheatsheet I made for `dplyr` join functions (not relevant yet but soon)