CrashCourse::Dplyr

Tinashe M. Tapera

Agenda

1. The Philosophy
2. The Basics
3. The Extras

The Philosophy

"`dplyr` is a grammar of data manipulation, providing a consistent set of **verbs** that help you solve the most common data manipulation challenges."

— https://dplyr.tidyverse.org/

- If `R` is a language, `dplyr` is a dialect
- Main focus on data munging within the R ecosystem:
  - You're gonna wanna use `tibble()`s
- Focuses on elegance, readability, parsimony, and reproducibility
- Part of the `tidyverse`, so works well with all of their packages

3 / 28

The Philosophy

- `dplyr` abstracts base R; does not replace direct knowledge
- Not very widely scoped; wouldn't use it for out-of-`tibble()` situations (but many situations in R can be manipulated into `tibble()`-friendly ones)
- Elegant, but not the fastest; with very large datasets, `data.table` is faster (source)

The Basics

"I claim that most single table problems can be solved with just five key verbs: filter, select, mutate, arrange and summarise, along with a 'by group' adverb."

— Hadley Wickham

The Basics | `%>%`

Pipes conjoin each `dplyr` verb by saying "and then...".

```
library(dplyr, warn.conflicts = FALSE, quietly = TRUE)
```

```
## Warning: package 'dplyr' was built under R version 3.5.1
```

```
iris %>%
  head() #note the indentation for readability
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

The Basics | `%>%`

- Typically, you could just do `head(iris)`: parsimonious & readable!
- What if you had 3, 4, or more functions wrapping around one object?

e.g. what are the coefficients of a linear model predicting Species in iris, using only rows where Sepal.Length is greater than the mean Sepal Length?

```
coef(lm(Species ~., data=subset(iris, iris$Sepal.Length > mean(iris$S
```

```
##  (Intercept) Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##    0.4760524   -0.2514633   -0.2395932    0.4583072    0.6171173
```

- Some programmers suggest breaking up their compound lines by assigning outputs to variables incrementally, e.g.

```
meanSepal = mean(iris$Sepal.Length)
subsdf = subset(iris, iris$Sepal.Length > meanSepal)
# etc...
```

- Really, tho...?

The Basics | `%>%`

In dplyr, it looks like this:

```
iris%>%                                          # the noun
  filter(Sepal.Length > mean(Sepal.Length))%>%   # first verb
  lm(Species ~ ., data=.)%>%                     # second verb
  coef()                                         # last verb
```

```
##  (Intercept) Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##    0.4760524   -0.2514633   -0.2395932    0.4583072    0.6171173
```

The Basics | .

- Formulas in R can make use of a period or dot operator: `lm(Species ~ ., data=iris)`. The dot refers to "all variables except those on the LHS/RHS".
- In `dplyr` (really, `magrittr`), the dot refers to the noun being passed around, and is implicit by default.
- It is described as a "dummy parameter" or "placeholder"

```
iris%>%                          # iris%>%
  head(2)                        #   head(x=.,n=2)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
```

- There are some nuances to its use that will come up as you get into more advanced operations

The Basics | `select()`

- Most important verb to understand
- For selecting columns or variables (as long as object is a kind of dataframe)

```
iris%>%
  select(Sepal.Length, Petal.Length)%>%
  head(3)
```

```
##   Sepal.Length Petal.Length
## 1          5.1          1.4
## 2          4.9          1.4
## 3          4.7          1.3
```

- Compare with bracket `[,]` indexing:

```
head( iris[, grep("Length$", names(iris))], 3)
```

```
##   Sepal.Length Petal.Length
## 1          5.1          1.4
## 2          4.9          1.4
## 3          4.7          1.3
```

10 / 28

The Basics | `select()`

So many helper functions with `select()`!!!

- `-` to drop, `Var1:Var5` for a range, `c(...)` for vectors
- `starts_with()`, `ends_with()`, `contains()`, `matches("regular_expression")` for regular expressions
- `one_of(c(...))` for optional matching
- `everything()` for everything that's left

```
iris %>%
  select(ends_with("Length")) %>%
  head(3)
```

```
##   Sepal.Length Petal.Length
## 1          5.1          1.4
## 2          4.9          1.4
## 3          4.7          1.3
```

The Basics | `filter()`

- For selecting rows of a dataframe

```
iris%>%
  filter(Species == "setosa")%>%
  head(3)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
```

- Compare with `which()`

```
head( iris[which(iris$Species == "setosa"), ] ,3)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
```

The Basics | `mutate()`

- Use `mutate()` to add new columns onto the dataframe (and use `transmute()` to only return the new column)
- Implicitly calls `select()`

```
iris %>%
  mutate(Sepal.Area = Sepal.Length*Sepal.Width) %>%
  head(3)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species Sepal.Area
## 1          5.1         3.5          1.4         0.2  setosa      17.85
## 2          4.9         3.0          1.4         0.2  setosa      14.70
## 3          4.7         3.2          1.3         0.2  setosa      15.04
```

The Basics | `arrange()`

- Sort by variable(s)

```
iris %>%
  arrange(-Sepal.Length, -Sepal.Width) %>%
  head(3)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 1          7.9         3.8          6.4         2.0 virginica
## 2          7.7         3.8          6.7         2.2 virginica
## 3          7.7         3.0          6.1         2.3 virginica
```

- Also respects grouping variables, too!

The Basics | `summarise()`

- Takes a range of rows and applies some function to return a dataframe of one value:

```
iris %>%
  summarise(my_mean=mean(Petal.Length))
```

```
##   my_mean
## 1   3.758
```

The Basics | `group_by()`

- Applies a grouping arrangement to be passed on to other functions further on
- All basic functions in `dplyr` respect grouping

```
iris %>%
  group_by(Species) %>%
  summarise(my_mean=mean(Petal.Length))
```

```
## # A tibble: 3 x 2
##   Species    my_mean
##   <fct>        <dbl>
## 1 setosa        1.46
## 2 versicolor    4.26
## 3 virginica     5.55
```

- You can group by as many variables as you'd like

The Basics | `gather()`

- `gather()` is great for turning wide-form data into long-form. Args as follows:

```
gather(key = name the variable stack,
    value = name the values,
    ... = which variables to gather (use select() helpers))
```

```
library(tidyr)
iris %>%
  gather(Petal_metric, value, starts_with("Petal")) %>%
  head(10)
```

```
##    Sepal.Length Sepal.Width Species Petal_metric value
## 1           5.1         3.5  setosa Petal.Length   1.4
## 2           4.9         3.0  setosa Petal.Length   1.4
## 3           4.7         3.2  setosa Petal.Length   1.3
## 4           4.6         3.1  setosa Petal.Length   1.5
## 5           5.0         3.6  setosa Petal.Length   1.4
## 6           5.4         3.9  setosa Petal.Length   1.7
## 7           4.6         3.4  setosa Petal.Length   1.4
## 8           5.0         3.4  setosa Petal.Length   1.5
## 9           4.4         2.9  setosa Petal.Length   1.4
## 10          4.9         3.1  setosa Petal.Length   1.5
```

17 / 28

The Basics | `spread()`

- The complement to `gather()`

```
spread(key = the variable to unstack,
    value = the variable with your stacked values)
```

```r
library(tibble)
iris %>%
  rownames_to_column("index") %>%                              # ?!?!
  gather(Petal_metric, value, starts_with("Petal")) %>%
  spread(Petal_metric, value) %>%
  arrange(as.numeric(index)) %>%                               # ?!?!
  head(5)
```

```
##   index Sepal.Length Sepal.Width Species Petal.Length Petal.Width
## 1     1          5.1         3.5  setosa          1.4         0.2
## 2     2          4.9         3.0  setosa          1.4         0.2
## 3     3          4.7         3.2  setosa          1.3         0.2
## 4     4          4.6         3.1  setosa          1.5         0.2
## 5     5          5.0         3.6  setosa          1.4         0.2
```

- ?!?! One quirk: `spread()` needs specific row indeces to unravel its values; throw in an explicit column index and order

The Basics | Other Common Functions

- `separate()`/`unite()`
    - Complementary string column "split" and "concatenate"
- `*_join()`
    - Traditional SQL-style joins (but with a nicer interface than `merge()`,`sqldf`, etc.)
- `sample_n()`/`sample_frac()`
    - Sampling rows of a dataframe (with or without replacement)
    - Much clearer than `iris[ sample(nrow(iris), n), ]`
- `slice()`
    - Positional row indexing
- Remember to `ungroup()` explicitly!
- Remember to use `rowwise()` to iterate (like calling `apply(MARGIN=2)`), because R does not like iterating rows naturally; it's vectorised!

The Extras | list-columns

- You can `nest()` dataframes and lists in `dplyr` to create list-columns

```
iris%>%
  nest(-Species)%>%
  as_tibble()            # for viewing on this slide
```

```
## # A tibble: 3 x 2
##   Species    data
##   <fct>      <list>
## 1 setosa     <data.frame [50 × 4]>
## 2 versicolor <data.frame [50 × 4]>
## 3 virginica  <data.frame [50 × 4]>
```

- You can then map operations on the list's objects with the `purrr` package (that's a topic for another day, though)

The Extras | list-columns

- Remember `summarise()`? It only works if the return of the summary gives you a single value vector. Using list-columns can help us override this
- What are the quantiles of `Sepal.Length` for each species of iris?

```
#base R, you'd have to call this three times for each species
quantile( iris[which(iris$Species == "setosa"), "Sepal.Length"] )
```

```
##   0%  25%  50%  75% 100% 
##  4.3  4.8  5.0  5.2  5.8
```

```
#dplyr without list-columns returns an error

iris %>%
  group_by(Species) %>%
  summarise(quant=quantile(Sepal.Length))
```

```
## Error in summarise_impl(.data, dots): Column `quant` must be length 1 (a s
```

The Extras | list-columns

```
# instead, just coerce the return value into a list-column
iris %>%
  group_by(Species) %>%
  summarise(Sepal_Length_quants = list(quantile(Sepal.Length)))
```

```
## # A tibble: 3 x 2
##   Species    Sepal_Length_quants
##   <fct>      <list>
## 1 setosa     <dbl [5]>
## 2 versicolor <dbl [5]>
## 3 virginica  <dbl [5]>
```

The Extras | list-columns

```
iris %>%
  group_by(Species) %>%
  summarise(Sepal_Length_quants = list(quantile(Sepal.Length))) %>%
  .$Sepal_Length_quants
```

```
## [[1]]
##   0%  25%  50%  75% 100%
##  4.3  4.8  5.0  5.2  5.8
##
## [[2]]
##   0%  25%  50%  75% 100%
##  4.9  5.6  5.9  6.3  7.0
##
## [[3]]
##    0%   25%   50%   75%  100%
## 4.900 6.225 6.500 6.900 7.900
```

The Extras | Scoped `filter_*()`

- You can create complex filtering conditions using scoped filters like `filter_at()`, `filter_all()`, and `filter_if()`
- These will return the rows of a dataframe once filtered on the specific variable predicates

e.g. let's filter only length variables from iris, where the length is greater than 5 for either of them

```
iris %>%
  filter_at(.vars = vars(contains("Length")),
            .vars_predicate = any_vars(. > 5)) %>%
  head(3)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          5.4         3.9          1.7         0.4  setosa
## 3          5.4         3.7          1.5         0.2  setosa
```

- Note the use of `vars()`, which explicitly calls `select()` helpers

24 / 28

The Extras | Scoped `mutate_*()`

- Similarly, there are scoped mutate and summarise calls

e.g. multiply only numeric variables by 2

```
iris %>%
  transmute_if(is.numeric,
               funs(new = . * 2)) %>%
  head(3)
```

```
##   Sepal.Length_new Sepal.Width_new Petal.Length_new Petal.Width_new
## 1             10.2             7.0              2.8             0.4
## 2              9.8             6.0              2.8             0.4
## 3              9.4             6.4              2.6             0.4
```

The Extras | Scoped `summarise_*()`

e.g. Let's summarise only width variables to get their means

```
# admittedly, this is overcomplicated
iris %>%
  summarise_at(vars(ends_with("width")),
               funs(mean)) %>%
  head(3)
```

```
##   Sepal.Width Petal.Width
## 1    3.057333    1.199333
```

- Note the use of `funs()`, which can accept any number of custom functions

Conclusion

- `dplyr` makes data munging cleaner and more interpretable
- There are lots of useful hidden functions under the hood
- Doesn't replace knowledge of base R
- Doesn't scale to HPC scenarios; best for making table summarisations and operations easier
- Huge community support means that you can figure out pretty much anything eventually

Thank you!