Introduction to R-Part 3

dplyr base R

Yebelay Berehan

yebelay.ma@gmail.com

2022-05-21

- Compares dplyr functions to their base R equivalents.
- This helps those familiar with base R understand better what dplyr does, and / or
- shows dplyr users how you might express the same ideas in base R code.

1. In dplyr you don't need to use `$` to refer to columns in the "current" data frame. This behaviour is inspired by the base functions `subset()` and `transform()`.
2. dplyr solutions tend to use a variety of single purpose verbs, while base R solutions typically tend to use `[` in a variety of ways, depending on the task at hand.
3. Multiple dplyr verbs are often strung together into a pipeline by `%>%`. In base R, you'll typically save intermediate results to a variable that you either discard, or repeatedly overwrite.

Compare dplyr vs base

- For one table verbs

dplyr

1. arrange(df, x)
2. distinct(df, x)
3. filter(df, x)
4. mutate(df, z = x + y)
5. pull(df, 1)
6. pull(df, x)
7. rename(df, y = x)
8. relocate(df, y)
9. select(df, x, y)
10. select(df, starts_with("x"))
11. summarise(df, mean(x))
12. slice(df, c(1, 2, 5))

base

1. df[order(x), , drop = FALSE]
2. df[!duplicated(x),,drop=FALSE] , unique()
3. df[which(x), , drop = FALSE] , subset()
4. df$z <- df$x + df$y , transform()
5. df[[1]]
6. df$x
7. names(df)[names(df)== "x"]<- "y"
8. df[union("y", names(df))]
9. df[c("x", "y")], subset()
10. df[grepl(names(df), "^x")]
11. mean(df$x) , tapply() , aggregate() , by()
12. df[c(1, 2, 5), , drop = FALSE]

arrange():

- Arrange rows by variables

`dplyr::arrange()` orders the rows of a data frame by the values of one or more columns:

```
library(dplyr)
mtcars <- as_tibble(mtcars)
iris <- as_tibble(iris)
```

```
mtcars %>% arrange(cyl, disp)
```

The `desc()` helper allows you to order selected variables in descending order:

```
mtcars %>% arrange(desc(cyl), desc(disp))
```

We can replicate in base R by using `[` with `order()`:

```
mtcars[order(mtcars$cyl, mtcars$disp), , drop = FALSE]
```

- Note the use of `drop = FALSE` is to get dataframe output in a data frame with a single column.
- Base R does not provide a convenient and general way to sort individual variables in descending order, so you have two options:
- For numeric variables, you can use `-x`.
- You can request `order()` to sort all variables in descending order.

```
mtcars[order(mtcars$cyl, mtcars$disp, decreasing = TRUE), , drop = FALSE]
mtcars[order(-mtcars$cyl, -mtcars$disp), , drop = FALSE]
```

`distinct()`:

- Select distinct/unique rows

`dplyr::distinct()` selects unique rows:

```
df <- tibble(
  x = sample(10, 100, rep = TRUE),
  y = sample(10, 100, rep = TRUE))
df %>% distinct(x) # selected columns
df %>% distinct(x, .keep_all = TRUE) # whole data frame
```

- There are two equivalents in base R, depending on whether you want the whole data frame, or just selected variables:

```
unique(df["x"]) # selected columns
df[!duplicated(df$x), , drop = FALSE] # whole data frame
```

`filter()`:

- Return rows with matching conditions
- `dplyr::filter()` selects rows where an expression is `TRUE`:

```
starwars %>% filter(species == "Human")
starwars %>% filter(mass > 1000)
starwars %>% filter(hair_color == "none" & eye_color == "black")
```

- The closest base equivalent (and the inspiration for `filter()`) is `subset()`:

```
subset(starwars, species == "Human")
subset(starwars, mass > 1000)
subset(starwars, hair_color == "none" & eye_color == "black")
```

- You can also use `[` but this also requires the use of `which()` to remove `NA`s:

```
starwars[which(starwars$species == "Human"), , drop = FALSE]
```

`mutate()`:

- Create or transform variables
- `dplyr::mutate()` creates new variables from existing variables:

```
df %>% mutate(z = x + y, z2 = z ^ 2)
```

- The closest base equivalent is `transform()`, but note that it cannot use freshly created variables:

```
head(transform(df, z = x + y, z2 = (x + y) ^ 2))
```

Alternatively, you can use `$<-`:

```
mtcars$cyl2 <- mtcars$cyl * 2
mtcars$cyl4 <- mtcars$cyl2 * 2
```

When applied to a grouped data frame, `dplyr::mutate()` computes new variable once per group:

```
gf <- tibble(g = c(1, 1, 2, 2), x = c(0.5, 1.5, 2.5, 3.5))
gf %>% group_by(g) %>%
  mutate(x_mean = mean(x), x_rank = rank(x))
```

To replicate this in base R, you can use `ave()`:

```
transform(gf,
  x_mean = ave(x, g, FUN = mean),
  x_rank = ave(x, g, FUN = rank))
```

pull():

- Pull out a single variable
- `dplyr::pull()` extracts a variable either by name or position:

```
mtcars %>% pull(1)
mtcars %>% pull(cyl)
```

- This equivalent to `[[` for positions and `$` for names:

```
mtcars[[1]]
mtcars$cyl
```

relocate():

- Change column order
- `dplyr::relocate()` makes it easy to move a set of columns to a new position (by default, the front):

```
# to front
mtcars %>% relocate(gear, carb)
# to back
mtcars %>% relocate(mpg, cyl, .after = last_col())
```

- We can replicate this in base R with a little set manipulation:

```
mtcars[union(c("gear", "carb"), names(mtcars))]
to_back <- c("mpg", "cyl")
mtcars[c(setdiff(names(mtcars), to_back), to_back)]
```

Moving columns to somewhere in the middle requires a little more set twiddling.

`rename()`:

- Rename variables by name
- `dplyr::rename()` allows you to rename variables by name or position:

```
iris %>% rename(sepal_length = Sepal.Length, sepal_width = 2)
```

- Renaming variables by position is straight forward in base R:

```
iris2 <- iris
names(iris2)[2] <- "sepal_width"
```

- Renaming variables by name requires a bit more work:

```
names(iris2)[names(iris2) == "Sepal.Length"] <- "sepal_length"
```

`rename_with()`:

- Rename variables with a function
- `dplyr::rename_with()` transform column names with a function:

```
iris %>% rename_with(toupper)
```

- A similar effect can be achieved with `setNames()` in base R:

```
setNames(iris, toupper(names(iris)))
```

`select()`:

- Select variables by name
- `dplyr::select()` subsets columns by position, name, function of name, or other property:

```
iris %>% select(1:3)
iris %>% select(Species, Sepal.Length)
iris %>% select(starts_with("Petal"))
iris %>% select(where(is.factor))
```

- Subsetting variables by position is straightforward in base R:

```
iris[1:3] # single argument selects columns; never drops
iris[1:3, , drop = FALSE]
```

- You have two options to subset by name:

```
iris[c("Species", "Sepal.Length")]
subset(iris, select = c(Species, Sepal.Length))
```

Subsetting by function of name requires a bit of work with `grep()`:

```
iris[grep("^Petal", names(iris))]
```

And you can use `Filter()` to subset by type:

```
Filter(is.factor, iris)
```

`summarise()`:

- Reduce multiple values down to a single value
- `dplyr::summarise()` computes one or more summaries for each group:

```
mtcars %>%
  group_by(cyl) %>%
  summarise(mean = mean(disp), n = n())
```

- I think the closest base R equivalent uses `by()`.
- Unfortunately `by()` returns a list of data frames, but you can combine them back together again with `do.call()` and `rbind()`:

```
mtcars_by <- by(mtcars, mtcars$cyl, function(df) {
  with(df, data.frame(cyl = cyl[[1]], mean = mean(disp), n = nrow(df)))
})
do.call(rbind, mtcars_by)
```

- `aggregate()` comes very close to providing an elegant answer:

```
agg <- aggregate(disp ~ cyl, mtcars, function(x) c(mean = mean(x), n = length(x)))
```

- But unfortunately while it looks like there are `disp.mean` and `disp.n` columns, it's actually a single matrix column:

```
str(agg)
```

You can see a variety of other options at https://gist.github.com/hadley/c430501804349d382ce90754936ab8ec.

`slice()`:

- Choose rows by position
- `slice()` selects rows with their location:

```
slice(mtcars, 25:n())
```

- This is straightforward to replicate with `[`:

```
mtcars[25:nrow(mtcars), , drop = FALSE]
```

Merging datasets

- When we want to merge two data frames, `x` and `y`), we have a variety of different ways to bring them together.
- Various base R `merge()` calls are replaced by a variety of dplyr `join()` functions.

dplyr

1. **inner_join(df1, df2)**
2. **left_join(df1, df2)**
3. **right_join(df1, df2)**
4. **full_join(df1, df2)**
5. **semi_join(df1, df2)**
6. **anti_join(df1, df2)**

base

1. **merge(df1, df2)**
2. **merge(df1, df2, all.x = TRUE)**
3. **merge(df1, df2, all.y = TRUE)**
4. **merge(df1, df2, all = TRUE)**
5. **df1[df1$x %in% df2$x, ,drop=FALSE]**
6. **df1[!df1$x %in% df2$x, ,drop=FALSE]**

Mutating joins

- dplyr's `inner_join()`, `left_join()`, `right_join()`, and `full_join()` add new columns from `y` to `x`, matching rows based on a set of "keys", and differ only in how missing matches are handled.
- They are equivalent to calls to `merge()` with various settings of the `all`, `all.x`, and `all.y` arguments.
- The main difference is the order of the rows:
- dplyr preserves the order of the `x` data frame.
- `merge()` sorts the key columns.

Filtering joins

- dplyr's `semi_join()` and `anti_join()` affect only the rows, not the columns:

```
band_members %>% semi_join(band_instruments)
band_members %>% anti_join(band_instruments)
```

- They can be replicated in base R with `[` and `%in%`:

```
band_members[band_members$name %in% band_instruments$name, , drop = FALSE]
band_members[!band_members$name %in% band_instruments$name, , drop = FALSE]
```

- Semi and anti joins with multiple key variables are considerably more challenging to implement.