dplyr <-> base R

This vignette compares dplyr functions to their base R equivalents. This helps those familiar with base R understand better what dplyr does, and shows dplyr users how you might express the same ideas in base R code. We'll start with a rough overview of the major differences, then discuss the one table verbs in more detail, followed by the two table verbs.

Overview

1. The code dplyr verbs input and output data frames. This contrasts with base R functions which more frequently work with individual vectors.
2. dplyr relies heavily on "non-standard evaluation" so that 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()`.
3. 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.
4. 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.
5. All dplyr verbs handle "grouped" data frames so that the code to perform a computation per-group looks very similar to code that works on a whole data frame. In base R, per-group operations tend to have varied forms.

One table verbs

The following table shows a condensed translation between dplyr verbs and their base R equivalents. The following sections describe each operation in more detail. You'll learn more about the dplyr verbs in their documentation and in `vignette("dplyr")`.

dplyr	base
`arrange(df, x)`	`df[order(x), , drop = FALSE]`
`distinct(df, x)`	`df[!duplicated(x), , drop = FALSE]`, `unique()`
`filter(df, x)`	`df[which(x), , drop = FALSE]`, `subset()`
`mutate(df, z = x + y)`	`df$z <- df$x + df$y`, `transform()`
`pull(df, 1)`	`df[[1]]`
`pull(df, x)`	`df$x`
`rename(df, y = x)`	`names(df)[names(df) == "x"] <- "y"`
`relocate(df, y)`	`df[union("y", names(df))]`
`select(df, x, y)`	`df[c("x", "y")]`, `subset()`
`select(df, starts_with("x"))`	`df[grepl("^x", names(df))]`
`summarise(df, mean(x))`	`mean(df$x)`, `tapply()`, `aggregate()`, `by()`

dplyr	base
`slice(df, c(1, 2, 5))`	`df[c(1, 2, 5), , drop = FALSE]`

To begin, we'll load dplyr and convert `mtcars` and `iris` to tibbles so that we can easily show only abbreviated output for each operation.

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

`arrange()`: Arrange rows by variables

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

```
mtcars %>% arrange(cyl, disp)
#> # A tibble: 32 × 11
#>     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
#>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1  33.9     4  71.1    65  4.22  1.84  19.9     1     1     4     1
#> 2  30.4     4  75.7    52  4.93  1.62  18.5     1     1     4     2
#> 3  32.4     4  78.7    66  4.08  2.2   19.5     1     1     4     1
#> 4  27.3     4  79      66  4.08  1.94  18.9     1     1     4     1
#> # ℹ 28 more rows
```

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

```
mtcars %>% arrange(desc(cyl), desc(disp))
#> # A tibble: 32 × 11
#>     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
#>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1  10.4     8   472   205  2.93  5.25  18.0     0     0     3     4
#> 2  10.4     8   460   215  3     5.42  17.8     0     0     3     4
#> 3  14.7     8   440   230  3.23  5.34  17.4     0     0     3     4
#> 4  19.2     8   400   175  3.08  3.84  17.0     0     0     3     2
#> # ℹ 28 more rows
```

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

```
mtcars[order(mtcars$cyl, mtcars$disp), , drop = FALSE]
#> # A tibble: 32 × 11
#>     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
#>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1  33.9     4  71.1    65  4.22  1.84  19.9     1     1     4     1
#> 2  30.4     4  75.7    52  4.93  1.62  18.5     1     1     4     2
#> 3  32.4     4  78.7    66  4.08  2.2   19.5     1     1     4     1
#> 4  27.3     4  79      66  4.08  1.94  18.9     1     1     4     1
#> # ℹ 28 more rows
```

Note the use of `drop = FALSE`. If you forget this, and the input is a data frame with a single column, the output will be a vector, not a data frame. This is a source of subtle bugs.

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
#> # A tibble: 10 × 1
#>       x
#>   <int>
#> 1     3
#> 2     5
#> 3     4
#> 4     7
#> # ℹ 6 more rows
df %>% distinct(x, .keep_all = TRUE) # whole data frame
#> # A tibble: 10 × 2
#>       x     y
#>   <int> <int>
#> 1     3     6
#> 2     5     2
#> 3     4     1
#> 4     7     1
#> # ℹ 6 more rows
```

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
#> # A tibble: 10 × 1
#>       x
#>   <int>
#> 1     3
#> 2     5
#> 3     4
#> 4     7
#> # ℹ 6 more rows
df[!duplicated(df$x), , drop = FALSE] # whole data frame
#> # A tibble: 10 × 2
#>       x     y
#>   <int> <int>
#> 1     3     6
#> 2     5     2
#> 3     4     1
#> 4     7     1
#> # ℹ 6 more rows
```

`filter()`: Return rows with matching conditions

`dplyr::filter()` selects rows where an expression is `TRUE`:

```
starwars %>% filter(species == "Human")
#> # A tibble: 35 × 14
#>   name      height  mass hair_color skin_color eye_color birth_year sex   gender
#>   <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
#> 1 Luke Sky…    172    77 blond      fair       blue            19   male  mascu…
#> 2 Darth Va…    202   136 none       white      yellow          41.9 male  mascu…
#> 3 Leia Org…    150    49 brown      light      brown           19   fema… femin…
#> 4 Owen Lars    178   120 brown, gr… light      blue            52   male  mascu…
#> # ℹ 31 more rows
#> # ℹ 5 more variables: homeworld <chr>, species <chr>, films <list>,
#> #   vehicles <list>, starships <list>
starwars %>% filter(mass > 1000)
#> # A tibble: 1 × 14
#>   name      height  mass hair_color skin_color eye_color birth_year sex   gender
#>   <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
#> 1 Jabba De…    175  1358 <NA>       green-tan… orange           600 herm… mascu…
#> # ℹ 5 more variables: homeworld <chr>, species <chr>, films <list>,
#> #   vehicles <list>, starships <list>
starwars %>% filter(hair_color == "none" & eye_color == "black")
#> # A tibble: 9 × 14
#>   name      height  mass hair_color skin_color eye_color birth_year sex   gender
#>   <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
#> 1 Nien Nunb    160    68 none       grey       black             NA male  mascu…
#> 2 Gasgano      122    NA none       white, bl… black             NA male  mascu…
#> 3 Kit Fisto    196    87 none       green      black             NA male  mascu…
#> 4 Plo Koon     188    80 none       orange     black             22 male  mascu…
#> # ℹ 5 more rows
#> # ℹ 5 more variables: homeworld <chr>, species <chr>, films <list>,
#> #   vehicles <list>, starships <list>
```

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

```
subset(starwars, species == "Human")
#> # A tibble: 35 × 14
#>   name      height  mass hair_color skin_color eye_color birth_year sex   gender
#>   <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
#> 1 Luke Sky…    172    77 blond      fair       blue            19   male  mascu…
#> 2 Darth Va…    202   136 none       white      yellow          41.9 male  mascu…
#> 3 Leia Org…    150    49 brown      light      brown           19   fema… femin…
#> 4 Owen Lars    178   120 brown, gr… light      blue            52   male  mascu…
#> # ℹ 31 more rows
#> # ℹ 5 more variables: homeworld <chr>, species <chr>, films <list>,
#> #   vehicles <list>, starships <list>
subset(starwars, mass > 1000)
#> # A tibble: 1 × 14
#>   name      height  mass hair_color skin_color eye_color birth_year sex   gender
#>   <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
#> 1 Jabba De…    175  1358 <NA>       green-tan… orange           600 herm… mascu…
#> # ℹ 5 more variables: homeworld <chr>, species <chr>, films <list>,
#> #   vehicles <list>, starships <list>
subset(starwars, hair_color == "none" & eye_color == "black")
#> # A tibble: 9 × 14
#>   name      height  mass hair_color skin_color eye_color birth_year sex   gender
#>   <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
#> 1 Nien Nunb    160    68 none       grey       black             NA male  mascu…
#> 2 Gasgano      122    NA none       white, bl… black             NA male  mascu…
#> 3 Kit Fisto    196    87 none       green      black             NA male  mascu…
#> 4 Plo Koon     188    80 none       orange     black             22 male  mascu…
#> # ℹ 5 more rows
#> # ℹ 5 more variables: homeworld <chr>, species <chr>, films <list>,
#> #   vehicles <list>, starships <list>
```

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

```
starwars[which(starwars$species == "Human"), , drop = FALSE]
#> # A tibble: 35 × 14
#>   name      height  mass hair_color skin_color eye_color birth_year sex   gender
#>   <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
#> 1 Luke Sky…    172    77 blond      fair       blue            19   male  mascu…
#> 2 Darth Va…    202   136 none       white      yellow          41.9 male  mascu…
#> 3 Leia Org…    150    49 brown      light      brown           19   fema… femin…
#> 4 Owen Lars    178   120 brown, gr… light      blue            52   male  mascu…
#> # ℹ 31 more rows
#> # ℹ 5 more variables: homeworld <chr>, species <chr>, films <list>,
#> #   vehicles <list>, starships <list>
starwars[which(starwars$mass > 1000), , drop = FALSE]
#> # A tibble: 1 × 14
#>   name      height  mass hair_color skin_color eye_color birth_year sex   gender
#>   <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
#> 1 Jabba De…    175  1358 <NA>       green-tan… orange           600 herm… mascu…
#> # ℹ 5 more variables: homeworld <chr>, species <chr>, films <list>,
#> #   vehicles <list>, starships <list>
```

```
starwars[which(starwars$hair_color == "none" & starwars$eye_color == "black"), , drop =
FALSE]
#> # A tibble: 9 × 14
#>   name      height  mass hair_color skin_color eye_color birth_year sex   gender
#>   <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
#> 1 Nien Nunb    160    68 none       grey       black             NA male  mascu…
#> 2 Gasgano      122    NA none       white, bl… black             NA male  mascu…
#> 3 Kit Fisto    196    87 none       green      black             NA male  mascu…
#> 4 Plo Koon     188    80 none       orange     black             22 male  mascu…
#> # ℹ 5 more rows
#> # ℹ 5 more variables: homeworld <chr>, species <chr>, films <list>,
#> #   vehicles <list>, starships <list>
```

`mutate()`: Create or transform variables

`dplyr::mutate()` creates new variables from existing variables:

```
df %>% mutate(z = x + y, z2 = z ^ 2)
#> # A tibble: 100 × 4
#>       x     y     z    z2
#>   <int> <int> <int> <dbl>
#> 1     3     6     9    81
#> 2     5     2     7    49
#> 3     4     1     5    25
#> 4     7     1     8    64
#> # ℹ 96 more rows
```

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))
#>    x y  z  z2
#> 1  3 6  9  81
#> 2  5 2  7  49
#> 3  4 1  5  25
#> 4  7 1  8  64
#> 5 10 7 17 289
#> 6  7 3 10 100
```

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))
#> # A tibble: 4 × 4
#> # Groups:   g [2]
#>       g     x x_mean x_rank
#>   <dbl> <dbl>  <dbl>  <dbl>
#> 1     1   0.5      1      1
#> 2     1   1.5      1      2
#> 3     2   2.5      3      1
#> 4     2   3.5      3      2
```

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)
)
#>   g   x x_mean x_rank
#> 1 1 0.5      1      1
#> 2 1 1.5      1      2
#> 3 2 2.5      3      1
#> 4 2 3.5      3      2
```

`pull()`: Pull out a single variable

`dplyr::pull()` extracts a variable either by name or position:

```
mtcars %>% pull(1)
#>  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
#> [16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
#> [31] 15.0 21.4
mtcars %>% pull(cyl)
#>  [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4
```

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

```
mtcars[[1]]
#>  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
#> [16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
#> [31] 15.0 21.4
mtcars$cyl
#>  [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4
```

`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)
#> # A tibble: 32 × 13
#>    gear  carb   mpg   cyl  disp    hp  drat    wt  qsec    vs    am  cyl2  cyl4
#>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1     4     4  21       6   160   110  3.9   2.62  16.5     0     1    12    24
#> 2     4     4  21       6   160   110  3.9   2.88  17.0     0     1    12    24
#> 3     4     1  22.8     4   108    93  3.85  2.32  18.6     1     1     8    16
#> 4     3     1  21.4     6   258   110  3.08  3.22  19.4     1     0    12    24
#> # ℹ 28 more rows

# to back
mtcars %>% relocate(mpg, cyl, .after = last_col())
#> # A tibble: 32 × 13
#>    disp    hp  drat    wt  qsec    vs    am  gear  carb  cyl2  cyl4   mpg   cyl
#>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1   160   110  3.9   2.62  16.5     0     1     4     4    12    24  21       6
#> 2   160   110  3.9   2.88  17.0     0     1     4     4    12    24  21       6
#> 3   108    93  3.85  2.32  18.6     1     1     4     1     8    16  22.8     4
#> 4   258   110  3.08  3.22  19.4     1     0     3     1    12    24  21.4     6
#> # ℹ 28 more rows
```

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

```
mtcars[union(c("gear", "carb"), names(mtcars))]
#> # A tibble: 32 × 13
#>    gear  carb   mpg   cyl  disp    hp  drat    wt  qsec    vs    am  cyl2  cyl4
#>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1     4     4  21       6   160   110  3.9   2.62  16.5     0     1    12    24
#> 2     4     4  21       6   160   110  3.9   2.88  17.0     0     1    12    24
#> 3     4     1  22.8     4   108    93  3.85  2.32  18.6     1     1     8    16
#> 4     3     1  21.4     6   258   110  3.08  3.22  19.4     1     0    12    24
#> # ℹ 28 more rows

to_back <- c("mpg", "cyl")
mtcars[c(setdiff(names(mtcars), to_back), to_back)]
#> # A tibble: 32 × 13
#>    disp    hp  drat    wt  qsec    vs    am  gear  carb  cyl2  cyl4   mpg   cyl
#>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1   160   110  3.9   2.62  16.5     0     1     4     4    12    24  21       6
#> 2   160   110  3.9   2.88  17.0     0     1     4     4    12    24  21       6
#> 3   108    93  3.85  2.32  18.6     1     1     4     1     8    16  22.8     4
#> 4   258   110  3.08  3.22  19.4     1     0     3     1    12    24  21.4     6
#> # ℹ 28 more rows
```

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)
#> # A tibble: 150 × 5
#>   sepal_length sepal_width Petal.Length Petal.Width Species
#>          <dbl>       <dbl>        <dbl>       <dbl> <fct>  
#> 1          5.1         3.5          1.4         0.2 setosa 
#> 2          4.9         3            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 
#> # ℹ 146 more rows
```

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 tibble: 150 × 5
#>   SEPAL.LENGTH SEPAL.WIDTH PETAL.LENGTH PETAL.WIDTH SPECIES
#>          <dbl>       <dbl>        <dbl>       <dbl> <fct>  
#> 1          5.1         3.5          1.4         0.2 setosa 
#> 2          4.9         3            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 
#> # ℹ 146 more rows
```

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

```
setNames(iris, toupper(names(iris)))
#> # A tibble: 150 × 5
#>   SEPAL.LENGTH SEPAL.WIDTH PETAL.LENGTH PETAL.WIDTH SPECIES
#>          <dbl>       <dbl>        <dbl>       <dbl> <fct>  
#> 1          5.1         3.5          1.4         0.2 setosa 
#> 2          4.9         3            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 
#> # ℹ 146 more rows
```

`select()`: Select variables by name

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

```
iris %>% select(1:3)
#> # A tibble: 150 × 3
#>   Sepal.Length Sepal.Width Petal.Length
#>          <dbl>       <dbl>        <dbl>
#> 1          5.1         3.5          1.4
#> 2          4.9         3            1.4
#> 3          4.7         3.2          1.3
#> 4          4.6         3.1          1.5
#> # ℹ 146 more rows
iris %>% select(Species, Sepal.Length)
#> # A tibble: 150 × 2
#>   Species Sepal.Length
#>   <fct>          <dbl>
#> 1 setosa           5.1
#> 2 setosa           4.9
#> 3 setosa           4.7
#> 4 setosa           4.6
#> # ℹ 146 more rows
iris %>% select(starts_with("Petal"))
#> # A tibble: 150 × 2
#>   Petal.Length Petal.Width
#>          <dbl>       <dbl>
#> 1          1.4         0.2
#> 2          1.4         0.2
#> 3          1.3         0.2
#> 4          1.5         0.2
#> # ℹ 146 more rows
iris %>% select(where(is.factor))
#> # A tibble: 150 × 1
#>   Species
#>   <fct>
#> 1 setosa
#> 2 setosa
#> 3 setosa
#> 4 setosa
#> # ℹ 146 more rows
```

Subsetting variables by position is straightforward in base R:

```
iris[1:3] # single argument selects columns; never drops
#> # A tibble: 150 × 3
#>   Sepal.Length Sepal.Width Petal.Length
#>          <dbl>       <dbl>        <dbl>
#> 1          5.1         3.5          1.4
#> 2          4.9         3            1.4
#> 3          4.7         3.2          1.3
#> 4          4.6         3.1          1.5
#> # ℹ 146 more rows
iris[1:3, , drop = FALSE]
#> # A tibble: 3 × 5
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#>          <dbl>       <dbl>        <dbl>       <dbl> <fct>
#> 1          5.1         3.5          1.4         0.2 setosa
#> 2          4.9         3            1.4         0.2 setosa
#> 3          4.7         3.2          1.3         0.2 setosa
```

You have two options to subset by name:

```
iris[c("Species", "Sepal.Length")]
#> # A tibble: 150 × 2
#>   Species Sepal.Length
#>   <fct>          <dbl>
#> 1 setosa           5.1
#> 2 setosa           4.9
#> 3 setosa           4.7
#> 4 setosa           4.6
#> # ℹ 146 more rows
subset(iris, select = c(Species, Sepal.Length))
#> # A tibble: 150 × 2
#>   Species Sepal.Length
#>   <fct>          <dbl>
#> 1 setosa           5.1
#> 2 setosa           4.9
#> 3 setosa           4.7
#> 4 setosa           4.6
#> # ℹ 146 more rows
```

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

```
iris[grep("^Petal", names(iris))]
#> # A tibble: 150 × 2
#>   Petal.Length Petal.Width
#>          <dbl>       <dbl>
#> 1          1.4         0.2
#> 2          1.4         0.2
#> 3          1.3         0.2
#> 4          1.5         0.2
#> # ℹ 146 more rows
```

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

```
Filter(is.factor, iris)
#> # A tibble: 150 × 1
#>   Species
#>   <fct>
#> 1 setosa
#> 2 setosa
#> 3 setosa
#> 4 setosa
#> # ℹ 146 more rows
```

`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())
#> # A tibble: 3 × 3
#>     cyl  mean     n
#>   <dbl> <dbl> <int>
#> 1     4  105.    11
#> 2     6  183.     7
#> 3     8  353.    14
```

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)
#>   cyl     mean  n
#> 4   4 105.1364 11
#> 6   6 183.3143  7
#> 8   8 353.1000 14
```

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

```
agg <- aggregate(disp ~ cyl, mtcars, function(x) c(mean = mean(x), n = length(x)))
agg
#>   cyl disp.mean   disp.n
#> 1   4  105.1364  11.0000
#> 2   6  183.3143   7.0000
#> 3   8  353.1000  14.0000
```

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

```
str(agg)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ cyl : num  4 6 8
#>  $ disp: num [1:3, 1:2] 105 183 353 11 7 ...
#>   ..- attr(*, "dimnames")=List of 2
#>   .. ..$ : NULL
#>   .. ..$ : chr [1:2] "mean" "n"
```

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())
#> # A tibble: 8 × 13
#>     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb  cyl2  cyl4
#>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1  19.2     8 400     175  3.08  3.84  17.0     0     0     3     2    16    32
#> 2  27.3     4  79      66  4.08  1.94  18.9     1     1     4     1     8    16
#> 3  26       4 120.     91  4.43  2.14  16.7     0     1     5     2     8    16
#> 4  30.4     4  95.1   113  3.77  1.51  16.9     1     1     5     2     8    16
#> # ℹ 4 more rows
```

This is straightforward to replicate with `[`:

```
mtcars[25:nrow(mtcars), , drop = FALSE]
#> # A tibble: 8 × 13
#>     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb  cyl2  cyl4
#>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1  19.2     8 400     175  3.08  3.84  17.0     0     0     3     2    16    32
#> 2  27.3     4  79      66  4.08  1.94  18.9     1     1     4     1     8    16
#> 3  26       4 120.     91  4.43  2.14  16.7     0     1     5     2     8    16
#> 4  30.4     4  95.1   113  3.77  1.51  16.9     1     1     5     2     8    16
#> # ℹ 4 more rows
```

Two-table verbs

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	base
`inner_join(df1, df2)`	`merge(df1, df2)`
`left_join(df1, df2)`	`merge(df1, df2, all.x = TRUE)`
`right_join(df1, df2)`	`merge(df1, df2, all.y = TRUE)`
`full_join(df1, df2)`	`merge(df1, df2, all = TRUE)`
`semi_join(df1, df2)`	`df1[df1$x %in% df2$x, , drop = FALSE]`

dplyr	base
`anti_join(df1, df2)`	`df1[!df1$x %in% df2$x, , drop = FALSE]`

For more information about two-table verbs, see `vignette("two-table")`.

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)
#> Joining with `by = join_by(name)`
#> # A tibble: 2 × 2
#>   name  band
#>   <chr> <chr>
#> 1 John  Beatles
#> 2 Paul  Beatles
band_members %>% anti_join(band_instruments)
#> Joining with `by = join_by(name)`
#> # A tibble: 1 × 2
#>   name  band
#>   <chr> <chr>
#> 1 Mick  Stones
```

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

```
band_members[band_members$name %in% band_instruments$name, , drop = FALSE]
#> # A tibble: 2 × 2
#>   name  band
#>   <chr> <chr>
#> 1 John  Beatles
#> 2 Paul  Beatles
band_members[!band_members$name %in% band_instruments$name, , drop = FALSE]
#> # A tibble: 1 × 2
#>   name  band
#>   <chr> <chr>
#> 1 Mick  Stones
```

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