The [dplyr](https://dplyr.tidyverse.org/) package is an essential tool for manipulating data in R. The [“Introduction to dplyr”](https://cran.r-project.org/web/packages/dplyr/vignettes/dplyr.html) vignette gives a good overview of the common dplyr functions (list taken from the vignette itself):

* filter() to select cases based on their values.
* arrange() to reorder the cases.
* select() and rename() to select variables based on their names.
* mutate() and transmute() to add new variables that are functions of existing variables.
* summarise() to condense multiple values to a single value.
* sample\_n() and sample\_frac() to take random samples.

The [“Two-table verbs”](https://cran.r-project.org/web/packages/dplyr/vignettes/two-table.html) vignette gives a good introduction to using dplyr function for joining two tables together. The [“Window functions”](https://cran.r-project.org/web/packages/dplyr/vignettes/window-functions.html) vignette talks about, well, window functions, which are defined as functions which take n values and return n values (as opposed to aggregation functions, which take n values but return one value). The window functions mentioned there are (list taken from the vignette itself):

* Ranking and ordering functions: row\_number(), min\_rank(), dense\_rank(), cume\_dist(), percent\_rank(), and ntile(). These functions all take a vector to order by, and return various types of ranks.
* Offsets: lead() and lag() allow you to access the previous and next values in a vector, making it easy to compute differences and trends.
* Cumulative aggregates: cumsum(), cummin(), cummax() (from base R), and cumall(), cumany(), and cummean() (from dplyr).

However, dplyr comes with several other functions that are not mentioned in the vignettes (or at least, not at length). In this post I’ll talk about some of them. (For the full list of dplyr functions, see the [reference manual](https://cran.r-project.org/web/packages/dplyr/dplyr.pdf).)

* Counting functions: n() and n\_distinct()
* If-else functions: if\_else() and case\_when()
* Comparison functions: between() and near()
* Selecting specific elements based on position: nth(), first() and last()
* Selecting specific rows based on position/value: slice() and top\_n()
* Utilities: coalesce() and pull()

To illustrate these functions, I will use the flights dataset in the nycflights13 package.

library(tidyverse)

library(nycflights13)

data(flights)

**Counting functions: n() and n\_distinct()**

Ok, these aren’t exactly lesser known functions but they’re certainly useful. n() counts the number of rows in each group. For example, to count the number of flights in each month:

flights %>% group\_by(month) %>%

summarize(count = n())

# # A tibble: 12 x 2

# month count

#

# 1 1 27004

# 2 2 24951

# 3 3 28834

# 4 4 28330

# 5 5 28796

# 6 6 28243

# 7 7 29425

# 8 8 29327

# 9 9 27574

# 10 10 28889

# 11 11 27268

# 12 12 28135

n\_distinct() counts the number of unique values in each group. To count the number of distinct values of day in the dataset:

flights %>% summarize(cnt = n\_distinct(day))

# # A tibble: 1 x 1

# cnt

#

# 1 31

as we expect, since the longest month only has 31 days. We can also count the number of unique sets of values across columns. To count the number of distinct (month, day) values:

flights %>% summarize(cnt = n\_distinct(month,day))

# # A tibble: 1 x 1

# cnt

#

# 1 365

as expected (number of days in a year).

**If-else functions: if\_else() and case\_when()**

if\_else() returns a value which depends on whether a given condition is true or not. It works like the base ifelse() function, except that it is stricter in that the returned value must be of the same type, whether the given condition is true or not. if\_else() is commonly used to create new columns. For example, we could create a new column that is “United” if a flight was from United Airlines, “Other” otherwise:

flights %>%

transmute(carrier = carrier,

isUA = if\_else(carrier == "UA", "United", "Otherwise")) %>%

head()

# # A tibble: 6 x 2

# carrier isUA

#

# 1 UA United

# 2 UA United

# 3 AA Otherwise

# 4 B6 Otherwise

# 5 DL Otherwise

# 6 UA United

case\_when() is like if\_else() except that we have more than 2 possible values for the output. It is “the R equivalent of the SQL CASE WHEN statement”. If none of the cases match, an NA is returned. As with if\_else(), the returned values for all the cases must be of the same type.

Here is some code to return “United” if carrier is UA, “JetBlue” if it is B6, and “Other” for all other carriers (notice that the syntax within case\_when() is a little different from the usual, and how we use TRUE to catch all the cases that don’t meet the first two conditions):

flights %>%

transmute(carrier = carrier,

newCol = case\_when(

carrier == "UA" ~ "United",

carrier == "B6" ~ "JetBlue",

TRUE ~ "Otherwise"

)) %>%

head()

# # A tibble: 6 x 2

# carrier newCol

#

# 1 UA United

# 2 UA United

# 3 AA Otherwise

# 4 B6 JetBlue

# 5 DL Otherwise

# 6 UA United

**Comparison functions: between() and near()**

between(x, left, right) is a shortcut for x >= left & x <= right that has an efficient implementation. It makes for more concise code too, at the expense of being unable to tell if the endpoints are included or not unless one is familiar with the function.

near() is a safer way than using == to compare if two vectors of floating point numbers are equal element-wise. This is because it uses tolerance which is based on the machine’s double precision. Here is the example provided in near()‘s documentation:

sqrt(2) ^ 2 == 2

# [1] FALSE

near(sqrt(2) ^ 2, 2)

# [1] TRUE

**Selecting specific elements based on position: nth(), first() and last()**

nth(x, n) returns the nth element in the vector x. An optional vector to determine the order can be passed to the function as well. If n is negative, we count from the end of x (e.g. -2 will return the second last value in x). first() and last() are obvious special cases of nth().

What’s interesting is that these functions are actually wrappers around [[. This means that they can be used to pull out the nth element in a list as well!

**Selecting specific rows based on position/value: slice() and top\_n()**

slice() allows us to select certain rows based on their ordinal position (i.e. row number). This works especially well with grouped dataframes. For example, if we want the first row for each date:

flights %>% group\_by(month, day) %>%

slice(1) %>%

select(month, day, dep\_time)

# # A tibble: 365 x 3

# # Groups: month, day [365]

# month day dep\_time

#

# 1 1 1 517

# 2 1 2 42

# 3 1 3 32

# 4 1 4 25

# 5 1 5 14

# 6 1 6 16

# 7 1 7 49

# 8 1 8 454

# 9 1 9 2

# 10 1 10 3

# # … with 355 more rows

A simple modification gives us the first two rows for each date:

flights %>% group\_by(month, day) %>%

slice(1:2) %>%

select(month, day, dep\_time)

# # A tibble: 730 x 3

# # Groups: month, day [365]

# month day dep\_time

#

# 1 1 1 517

# 2 1 1 533

# 3 1 2 42

# 4 1 2 126

# 5 1 3 32

# 6 1 3 50

# 7 1 4 25

# 8 1 4 106

# 9 1 5 14

# 10 1 5 37

# # … with 720 more rows

In contrast to slice() which picks out rows by their position (within the group), top\_n() picks out rows by their value in a pre-specified column (within the group). For example, if we wanted to see the carriers for the top 3 longest departure delays for each day, we could use this code:

flights %>% group\_by(month, day) %>%

top\_n(3, dep\_delay) %>%

select(month, day, dep\_delay, carrier)

# # A tibble: 1,108 x 4

# # Groups: month, day [365]

# month day dep\_delay carrier

#

# 1 1 1 853 MQ

# 2 1 1 290 EV

# 3 1 1 379 EV

# 4 1 2 334 UA

# 5 1 2 337 AA

# 6 1 2 379 UA

# 7 1 3 268 DL

# 8 1 3 252 B6

# 9 1 3 291 9E

# 10 1 4 208 B6

# # … with 1,098 more rows

Notice that while we get the top 3 rows in dep\_delay for each day, the rows are not sorted according to the dep\_delay column. To get the bottom three rows instead, use a negative number:

flights %>% group\_by(month, day) %>%

top\_n(-3, dep\_delay) %>%

select(month, day, dep\_delay, carrier)

# # A tibble: 1,504 x 4

# # Groups: month, day [365]

# month day dep\_delay carrier

#

# 1 1 1 -15 MQ

# 2 1 1 -14 F9

# 3 1 1 -15 AA

# 4 1 2 -13 AA

# 5 1 2 -13 UA

# 6 1 2 -12 AA

# 7 1 2 -12 9E

# 8 1 3 -12 EV

# 9 1 3 -12 EV

# 10 1 3 -13 B6

# # … with 1,494 more rows

Notice how Jan 2nd has 4 entries, not 3? That’s because there was a tie for 3rd place. top\_n() either takes all rows with a value or none of them. Another gotcha: if the column that you are sorting against is not specified, it defaults to the last variable in the data frame, NOT to ordinal position/row number.

**Utilities: coalesce() and pull()**

From the documnetation: “Given a set of vectors, coalesce() finds the first non-missing value at each position. This is inspired by the SQL COALESCE function which does the same thing for NULLs.”

I haven’t had to use this function myself so far, but I imagine it could be useful when you have a number of different data sources reporting the same thing but with different missing values in each source. Here is a contrived example:

x <- c(NA, NA, 3)

y <- c(1, NA, 4)

z <- c(NA, 2, NA)

coalesce(x, y, z)

# [1] 1 2 3

Notice that the third value in y showing a 4 is ignored.

pull() works like [[: it allows you to pull out a particular column (as a vector). The variable you pass to the function can be a variable name, a positive integer (giving the column position when counting from the left) or a negative integer (giving the column position when counting from the right).