Introduction to Dplyr

This document gives an overview of many of the features of the dplyr library include in the “tidyverse” of related R packages. First we will load the library and a sample dataset.

```r
#install.packages("tidyverse")
library(tidyverse)
## Loading tidyverse: ggplot2
## Loading tidyverse: tibble
## Loading tidyverse: tidyr
## Loading tidyverse: readr
## Loading tidyverse: purrr
## Loading tidyverse: dplyr

## Conflicts with tidy packages ----------------------------------------------
## filter(): dplyr, stats
## lag(): dplyr, stats

#install.packages("nycflights13")
library(nycflights13)
# Show fewer rows by default in this document
options(tibble.print_min = 5L, tibble.print_max = 5L)

We will primarily be using the `flights` data:

```r
flights

## # A tibble: 336,776 x 19
## #  year month day dep_time sched.dep_time dep_delay arr_time
## # <int> <int> <int> <int> <int> <dbl> <int>
##  1 2013 1 1 517 515 2 830
##  2 2013 1 1 533 529 4 850
##  3 2013 1 1 542 540 2 923
##  4 2013 1 1 544 545 -1 1004
##  5 2013 1 1 554 600 -6 812
## ... with 336,771 more rows, and 12 more variables: sched.arr_time <int>,
## arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## minute <dbl>, time_hour <dttm>
```

Filtering Rows

Find all flights from Detroit in June (in 2013)

```r
# Same as using base R
# flights[flights$dest=="DTW" & flights$month==6, ]
# subset(flights, dest=="DTW" & month==6)
filter(flights, dest=="DTW" & month==6)
```
Selecting Columns

# List columns
select(flights, dep_time, arr_time, carrier)

# A tibble: 336,776 x 3
#  dep_time arr_time carrier
#   <int>   <int> <chr>
# 1    517    830 UA
# 2    533    850 UA
# 3    542    923 AA
# 4    544   1004 B6
# 5    554    812 DL
# ... with 336,771 more rows

# Exclude columns
select(flights, -year, -tailnum)

# A tibble: 336,776 x 17
#  month  day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#   <int> <int>   <int>       <int>      <dbl>   <int>       <int>
# 1     1     1     517          515       2     830          819
# 2     1     1     533          529       4     850          830
# 3     1     1     542          540       2     923          850
# 4     1     1     544          545      -1   1004         1022
# 5     1     1     554          600      -6     812          837
# ... with 336,771 more rows, and 10 more variables: arr_delay <dbl>,
# carrier <chr>, flight <int>, origin <chr>, dest <chr>, air_time <dbl>,
# distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>

# Select column range (in data.frame order)
select(flights, month:dep_delay)

# A tibble: 336,776 x 5
#  month  day dep_time sched_dep_time dep_delay
#   <int> <int>   <int>       <int>      <dbl>
# 1     1     1     517          515       2
## 2 1 1 533 529 4
## 3 1 1 542 540 2
## 4 1 1 544 545 -1
## 5 1 1 554 600 -6
## ... with 336,771 more rows

# Name starts with
```
select(flights, starts_with("d"))
```

## A tibble: 336,776 x 5
## day dep_time dep_delay dest distance
## <int> <int> <chr> <dbl>
## 1 1 517 2 IAH 1400
## 2 1 533 4 IAH 1416
## 3 1 542 2 MIA 1089
## 4 1 544 -1 BQN 1576
## 5 1 554 -6 ATL 762
## ... with 336,771 more rows

# Name ends with
```
select(flights, ends_with("time"))
```

## A tibble: 336,776 x 5
## dep_time sched_dep_time arr_time sched_arr_time air_time
## <int> <int> <int> <int> <dbl>
## 1 517 515 830 819 227
## 2 533 529 850 830 227
## 3 542 540 923 850 160
## 4 544 545 1004 1022 183
## 5 554 600 812 837 116
## ... with 336,771 more rows

# Name contains
```
select(flights, contains("arr"))
```

## A tibble: 336,776 x 4
## arr_time sched_arr_time arr_delay carrier
## <int> <int> <dbl> <chr>
## 1 830 819 11 UA
## 2 850 830 20 UA
## 3 923 850 33 AA
## 4 1004 1022 -18 B6
## 5 812 837 -25 DL
## ... with 336,771 more rows

# Name doesn’t start with
```
select(flights, -starts_with("d"))
```

## A tibble: 336,776 x 14
## year month sched_dep_time arr_time sched_arr_time arr_delay carrier
## <int> <int> <int> <int> <int> <dbl> <chr>
## 1 2013 1 515 830 819 11 UA
## 2 2013 1 529 850 830 20 UA
## 3 2013 1 540 923 850 33 AA
## 4 2013 1 545 1004 1022 -18 B6
## 5 2013 1 600 812 837 -25 DL
## ... with 336,771 more rows, and 7 more variables: flight <int>,

3
Look at the `?select` help page for a list of function to help you select multiple columns.

### Verb composition with pipes

Traditionally, we combine functions via nesting, which works but is hard to read

```r
select(filter(flights, dest="DTW"), carrier)
```

The `%>%` allows us to take an object, and pass it as the first parameter to another function. The above is the same as

```r
flights %>%
  filter(dest="DTW") %>%
  select(carrier)
```

You can unroll any function with this operator
### 1.62

```
.5 %>% sin() %>% exp %>% round(2)
```

### Sorting Data

Use `arrange()` to sort data. You just specify the column names you want to sort by, use `desc()` to reverse the sort order for a given column.

```r
flights %>% arrange(sched_dep_time)
```

```
## # A tibble: 336,776 x 19
## # [[year month day dep_time sched_dep_time dep_delay arr_time
## # <int> <int> <int> <int> <int> <dbl> <int>
## # 1 2013 7 27 NA 106 NA NA
## # 2 2013 1 2 458 500 -2 703
## # 3 2013 1 3 458 500 -2 650
## # 4 2013 1 4 456 500 -4 631
## # 5 2013 1 5 458 500 -2 640
## # ... with 336,771 more rows, and 12 more variables: sched_arr_time <int>,
## # arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## # origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## # minute <dbl>, time_hour <dttm>
```

```r
flights %>% arrange(month, desc(day))
```

```
## # A tibble: 336,776 x 19
## # [[year month day dep_time sched_dep_time dep_delay arr_time
## # <int> <int> <int> <int> <int> <dbl> <int>
## # 1 2013 1 31 1 2100 181 124
## # 2 2013 1 31 4 2359 5 455
## # 3 2013 1 31 7 2359 8 453
## # 4 2013 1 31 12 2250 82 132
## # 5 2013 1 31 26 2154 152 328
## # ... with 336,771 more rows, and 12 more variables: sched_arr_time <int>,
## # arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## # origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## # minute <dbl>, time_hour <dttm>
```

```r
flights %>% arrange(desc(dep_time - sched_dep_time ))
```

```
## # A tibble: 336,776 x 19
## # [[year month day dep_time sched_dep_time dep_delay arr_time
## # <int> <int> <int> <int> <int> <dbl> <int>
## # 1 2013 3 17 2321 810 911 135
## # 2 2013 7 22 2257 759 898 121
## # 3 2013 2 10 2243 830 853 100
## # 4 2013 2 19 2324 1016 788 114
## # 5 2013 2 24 1921 615 786 2135
## # ... with 336,771 more rows, and 12 more variables: sched_arr_time <int>,
## # arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## # origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
```
Creating New Variables

Use `mutate()` to create columns from existing columns or values

```r
flights %>%
  mutate(speed = distance / (air_time / 60)) %>%
  arrange(desc(speed)) %>%
  select(flight, speed)
```

```r
# A tibble: 336,776 x 2
#  flight   speed
#   <int>    <dbl>
# 1   1499  703.38
# 2   4667  650.32
# 3   4292  648.00
# 4   3805  641.14
# 5   1902  591.43
# ... with 336,771 more rows
```

You can create multiple columns by separating them with a comma; you can use any previously created columns as well

```r
flights %>%
  mutate(
    dist_km = distance * 1.61,
    hours = air_time / 60,
    kph = dist_km / hours)
```

```r
# A tibble: 336,776 x 2
#  flight   kph
#   <int>   <dbl>
# 1   1545  595.77
# 2   1714  602.58
# 3   1141  657.48
# 4    725  831.92
# 5    461  634.56
# ... with 336,771 more rows
```

Use `summarize()` to collapse observations (only keeps columns for which you specified a summarization strategy)

```r
flights %>%
  filter(!is.na(arr_delay)) %>%
  summarize(avg_arr_delay = mean(arr_delay))
```

```r
# A tibble: 1 x 1
#  avg_arr_delay
#       <dbl>
# 1      6.8954
```
Grouping Data

Perhaps the most powerful feature of dplyr is its grouping abilities. You can specify a column (or columns) for which mutate() and summarize() happen independently for each unique value in that column (or unique combination or values).

Using summarize() will reduce the total number of rows:

```r
flights %>%
  filter(!is.na(arr_delay)) %>%
  group_by(carrier) %>%
  summarize(avg_arr_delay = mean(arr_delay))
```

```
## # A tibble: 16 x 2
## # Groups: carrier [16]
## carrier avg_arr_delay
##  <chr>       <dbl>
## 1 9E         7.3796692
## 2 AA        0.3642909
## 3 AS      -9.9308886
## 4 B6        9.4579733
## 5 DL       1.6443409
## # ... with 11 more rows
```

Using mutate() will keep the same number of rows and won’t drop any columns:

```r
flights %>%
  filter(!is.na(arr_delay)) %>%
  group_by(carrier) %>%
  mutate(avg_arr_delay = mean(arr_delay)) %>%
  select(carrier, arr_delay, avg_arr_delay)
```

```
## # A tibble: 327,346 x 3
## # Groups: carrier [16]
## carrier       arr_delay avg_arr_delay
##  <chr>          <dbl>        <dbl>
## 1 UA            11^          3.5580111
## 2 UA            20          3.5580111
## 3 AA            33          0.3642909
## 4 B6          -18^         9.4579733
## 5 DL           -25          1.6443409
## # ... with 327,341 more rows
```

Joining data

When finding carriers with the largest flight delay, we were left with a carrier code rather than a carrier name; but who exactly is 9E?

```r
flights %>%
  filter(!is.na(arr_delay)) %>%
  group_by(carrier) %>%
  summarize(avg_arr_delay = mean(arr_delay))
```

```
## # A tibble: 16 x 2
## # Groups: carrier [16]
## carrier avg_arr_delay
##  <chr>       <dbl>
## 1 9E         7.3796692
```

7
There is another table that has a lookup from carrier code to carrier name called `airlines`.

```r
## A tibble: 16 x 2
##  carrier name
##  <chr> <chr>
## 1 9E   Endeavor Air Inc.
## 2 AA   American Airlines Inc.
## 3 AS   Alaska Airlines Inc.
## 4 B6   JetBlue Airways
## 5 DL   Delta Air Lines Inc.
## # ... with 11 more rows
```

We can use `left_join` to merge in the carrier name:

```r
flights %>%
  filter(!is.na(arr_delay)) %>%
  group_by(carrier) %>%
  summarize(avg_arr_delay = mean(arr_delay)) %>%
  left_join(airlines)

## Joining, by = "carrier"
## A tibble: 16 x 3
##  carrier avg_arr_delay name
##  <chr>       <dbl>     <chr>
## 1 9E          7.3796692 Endeavor Air Inc.
## 2 AA          0.3642909 American Airlines Inc.
## 3 AS         -9.9308886 Alaska Airlines Inc.
## 4 B6          9.4579733 JetBlue Airways
## 5 DL         1.6443409 Delta Air Lines Inc.
## # ... with 11 more rows
```

Here we use two sample tables `x` and `y` to demonstrate the other types of joins:

```r
x <- tribble(~key, ~xval, 1, "x1", 2, "x2", 3, "x3")
y <- tribble(~key, ~yval, 1, "y1", 2, "y2", 4, "y3")
inner_join(x, y)

## Joining, by = "key"
## A tibble: 2 x 3
##   key xval yval
##  <dbl> <chr> <chr>
```
## 1 1 x1 y1
## 2 2 x2 y2

```r
left_join(x, y)
```

## Joining, by = "key"
##  # A tibble: 3 x 3
##  key  xval  yval
##    <dbl> <chr> <chr>
## 1 1 x1 y1
## 2 2 x2 y2
## 3 3 x3 <NA>

```r
right_join(x, y)
```

## Joining, by = "key"
##  # A tibble: 3 x 3
##  key xval yval
##    <dbl> <chr> <chr>
## 1 1 x1 y1
## 2 2 x2 y2
## 3 4 <NA> y3

```r
full_join(x, y)
```

## Joining, by = "key"
##  # A tibble: 4 x 3
##  key xval yval
##    <dbl> <chr> <chr>
## 1 1 x1 y1
## 2 2 x2 y2
## 3 3 x3 <NA>
## 4 4 <NA> y3

And you can use non-merging joins to keep or drop rows that match keys from another table. Note that no new columns are added, just the rows of the input tables are filtered

```r
z <- tribble(~key, ~zval,
1, "z1",
3, "z2")
```

```r
semi_join(x, z)
```

## Joining, by = "key"
##  # A tibble: 2 x 2
##  key  xval
##    <chr>
## 1 x1
## 2 x3

```r
semi_join(y, z)
```

## Joining, by = "key"
##  # A tibble: 1 x 2
##  key  yval
##    <chr>
## 1 y2
## 1 1 y1

anti_join(x, z)

## Joining, by = "key"

## # A tibble: 1 x 2
## key xval
## <dbl> <chr>
## 1 2 x2

The join commands will join on all matching column names. You can more explicitly control this as well. The planes table has information about the aircraft used during the flight. It also happens to have a column named “year” indicating when the aircraft was built. When joining this data to flights, we only want to join on “tailnum” – not “tailnum” and “year”.

flights %>%
  inner_join(planes) %>%
  nrow() # wrong, only planes from 2013 are selected

## Joining, by = c("year", "tailnum")

## [1] 4630

flights %>%
  inner_join(planes, "tailnum") %>%
  nrow() # right

## [1] 284170

### Subsetting functions

distinct() will return unique combinations of column values and nothing else

flights %>%
distinct(tailnum, carrier)

## # A tibble: 4,067 x 2
## carrier tailnum
## <chr> <chr>
## 1 UA N14228
## 2 UA N24211
## 3 AA N619AA
## 4 B6 N804JB
## 5 DL N668DN
## # ... with 4,062 more rows

The count() is like distinct() except it also returns the number of times each value was observed. It’s basically a shortcut for group_by() %>% summarize(). For example

flights %>% count(carrier)

## # A tibble: 16 x 2
## carrier n
## <chr> <int>
## 1 9E 18460
## 2 AA 32729
## 3 AS 714
## 4 B6 54635
flights %>% group_by(carrier) %>% summarize(n=n())

flights %>% sample_n(3)

flights %>% sample_n(3)

You might also consider anti_join and semi_join to be subsetting commands rather than joining commands.

_at/_if/_all

The summarize(), mutate() and group_by() functions all have _all(), _at() and _if() variants that make it easier to apply the same function or functions to multiple columns.

mutate_at(), summarize_at() and group_by_at() allow you to choose columns in the same way you can do with select using the vars() helper function. This will take the mean of all columns that end in “_time”

flights %>%
summarize_at(vars(ends_with("time")), mean, na.rm=T)

## A tibble: 1 x 5
##    dep_time sched_dep_time arr_time sched_arr_time air_time

mutate_if(), summarize_if() and group_by_if() allow you run a function on each column to choose only columns that meet a certain criteria. This can use useful for extracting columns of a certain class. Note you can also apply more than one function to these columns if you use the `funs()` helper function. This example will calculate the mean and variance for all numeric columns.

```r
flights %>%
  mutate_if(is.numeric, funs(mean, var), na.rm=T)
```

The `_all()` versions of these functions will apply the same transformations to call non-grouped columns in the data source.

Other useful functions

The `lead()` and `lag()` functions are useful for selecting the next or previous values in a sequence (especially for time series data).

```r
x<-1:5
lead(x)
```

```r
## [1] 2 3 4 5 NA
```

```r
lag(x)
```

```r
## [1] NA 1 2 3 4
```

The `coalesce()` function will return the first non-missing value from the vectors you pass to it. This is useful when you have multiple columns where only one column contains a value and you want to collapse them to a single vector.

```r
coalesce(c(NA,2,NA), c(1, NA, NA), 3)
```

```r
## [1] 1 2 3
```

When using other `dplyr` verbs, the `n()` and `n_distinct()` functions will return the total number of observations or the number of unique observations respectively. In this example we look at the tail number for each plane to see how many total flights it took and also look at the number of distinct flight numbers that plane was a part of.

```r
flights %>%
  group_by(tailnum) %>%
  summarize(flights=n(), routes=n_distinct(flight))
```

```r
## # A tibble: 4,044 x 3
```
The `recode()` function allows you to swap out certain values in a vector with different values.

```r
recode(letters[1:5], b="boo")
```

## [1] "a" "boo" "c" "d" "e"

The `case_when()` function allows more complex transformations than `recode()`. It’s a good alternative to a bunch of nested `ifelse()` calls that you might need to use in base R. Each parameter should be a formula with a left-hand side value that evaluates to TRUE or FALSE and a right-hand side to return when that boolean value is TRUE. Only the value for the first TRUE is returned.

Here’s a classic example of the “fizz buzz” problem where you are supposed to return the numbers 1-50 but replace all those values divisible by 5 with “fizz” and the values divisible by 7 with “buzz” and those divisible by both 5 and 7 by “fizz buzz”

```r
x <- 1:50
case_when(
  x %% 5 == 0 ~ "fizz",
  x %% 7 == 0 ~ "buzz",
  TRUE ~ as.character(x)
)
```

## [1] "1" "2" "3" "4" "fizz"
## [6] "6" "buzz" "8" "9" "fizz"
## [11] "11" "12" "13" "buzz" "fizz"
## [16] "16" "17" "18" "19" "fizz"
## [21] "buzz" "22" "23" "24" "fizz"
## [26] "26" "27" "buzz" "29" "fizz"
## [31] "31" "32" "33" "34" "fizz buzz"
## [36] "36" "37" "38" "39" "fizz"
## [41] "41" "buzz" "43" "44" "fizz"
## [46] "46" "47" "48" "buzz" "fizz"

Combining data frames

The `bind_rows()` and `bind_columns()` functions are alternatives to the base functions `rbind()` and `cbind()` that are list-friendly. Many times you end up with data.frames in a list that you want to combine in a single data.frame. These functions can help.

In this example, we have a list of two tibbles. We can combine them with `bind_rows`

```r
x <- list(  
  data_frame(a=1:2, z=letters[1:2]),  
  data_frame(a=14:20, z=letters[14:20])  
)
bind_rows(x)
```

## # A tibble: 9 x 2
```
## a  z
##  <int> <chr>
##   1  1 a
##   2  2 b
##   3 14 n
##   4 15 o
##   5 16 p
## # ... with 4 more rows

bind_rows(x[[1]], x[[2]])
```

```
## A tibble: 9 x 2
##  a  z
## <int> <chr>
## 1 1 a
## 2 2 b
## 3 14 n
## 4 15 o
## 5 16 p
## # ... with 4 more rows
```

## Programming with dplyr

Since dplyr uses non-standard evaluation to allow you to specify data.frame column names without quotes, it can be tricky to write functions that use dply commands. Note that the first attempt at writing a function doesn't work

```
# Normal command, works fun
flights %>%
group_by(carrier) %>%
  summarize(delay=mean(arr_delay, na.rm=T))
```

```
## A tibble: 16 x 2
##  carrier delay
## <chr>  <dbl>
## 1 9E  7.3796692
## 2 AA  0.3642909
## 3 AS -9.9308886
## 4 B6  9.4579733
## 5 DL  1.6443409
## # ... with 11 more rows

# DOESN'T WORK
f <- function(x) {
  flights %>%
    group_by(x) %>%
    summarize(delay=mean(arr_delay, na.rm=T))
}
f(carrier)
```

```
## Error in grouped_df_impl(data, unname(vars), drop): Column `x` is unknown
```

The latest version of dplyr (0.7) introduced new way to write functions. Previously you would use the standard-evaluation version of functions that ended in an underscore (use `mutate_` rather than `mutate`); but the new version now uses “quosures” to allow you to pass column names. Here are two examples of functions that will work
```r
f <- function(x) {
  flights %>% group_by(x) %>%
  summarize(delay = mean(arr_delay, na.rm=T))
}
f(quosure(carrier))
```

```
## # A tibble: 16 x 2
##  carrier  delay
##  <chr> <dbl>
## 1 9E 7.3796692
## 2 AA 0.3642909
## 3 AS -9.9308886
## 4 B6 9.4579733
## 5 DL 1.6443409
## # ... with 11 more rows
```

```r
g <- function(x) {
  x <- enquo(x)
  flights %>% group_by(x) %>%
  summarize(delay = mean(arr_delay, na.rm=T))
}
g(carrier)
```

```
## # A tibble: 16 x 2
##  carrier delay
##  <chr> <dbl>
## 1 9E 7.3796692
## 2 AA 0.3642909
## 3 AS -9.9308886
## 4 B6 9.4579733
## 5 DL 1.6443409
## # ... with 11 more rows
```

We can either use `quosure()` to create our own quosure with the column name, or we can use `enquo()` to turn a function parameter into a quosure.

Finally, in base R it’s complicated to dynamically set the name of a parameter to a function (the name being the part to the left of the `=` in a call like `f(a=b)`). The latest dplyr functions now also allow you to use the value of a variable as a parameter name if you use `:=` rather than `=`. For example

```r
h <- function(x) {
  x <- enquo(x)
  outname <- paste(quosure_name(x), "delay", sep="_")
  flights %>% group_by(x) %>%
  summarize(!outname := mean(arr_delay, na.rm=T))
}
h(carrier)
```

```
## # A tibble: 16 x 2
##  carrier carrier_delay
##  <chr> <chr>
## 1 9E 9E delay
## 2 AA AA delay
## 3 AS AS delay
## 4 B6 B6 delay
## 5 DL DL delay
## # ... with 11 more rows
```

15
## # ... with 11 more rows