# 04 - Data Transformation

04-transform.pdf

# Required Packages and Data

```r
library(tidyverse)
library(nycflights13)
```

Remember, if you are getting the error:

```
> Error in library(nycflights13) : there is no
package called ‘nycflights13’
```

then you have not installed the `nycflights13` on your computer. You can do so by:

- `Tools -> Install Packages...` from RStudio.
- or, typing `install.packages("nycflights13")` in console or

# Practice

You need to **practice** to become proficient with the tools we are covering. The best way to do this is start analyzing data that is interesting to you. Here are some places:

- Many R packages have interesting data: `lahman`, `gapminder`, `acs`
- https://www.springboard.com/blog/free-public-data-sets-data-science-project/
- https://www.dataquest.io/blog/free-datasets-for-projects/

Look on-line and find something interests you. I can help you get the data into R if necessary, just ask.

# Data Transformation

# Working with data

When working with data you must:

1. Figure out what you want to do.
2. Precisely describe what you want to do in such a way that the compute can understand it (i.e. program it).
3. Execute the program.

The `dplyr` package makes some of these steps fast and easy:

- By constraining your options, it simplifies how you can think about common data manipulation tasks.
- It provides simple “verbs”, functions that correspond to the most common data manipulation tasks, to help you translate those thoughts into code.
- It uses efficient data storage backends, so you spend less time waiting for the computer.

# nycflights13

To explore the basic data manipulation verbs of `dplyr`, we'll use the `flights` data frame from the `nycflights13` package. This data frame contains all 336,776 flights that departed from New York City in 2013. The data comes from the US Bureau of Transportation Statistics, and is documented in `?nycflights13`.

# nycflights13

```
#- Load the flights data from nycflights13 package
library(nycflights13)
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
#> 6  2013     1     1      554            558        -4      740
#> # ... with 3.368e+05 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>
```

A `tibble` is a special data frame. See Chapter 10 of RDS for more details on the differences between `tibble` and `data.frame`.

# dyplr Package

# dyplr help

- Data Transformation Cheatsheet
- Introduction to the `dplyr` package

The functions in the `dplyr` package translate well to SQL functionality. In fact, you can run dplyr queries on a SQL data base (https://db.rstudio.com/dplyr/) and bypass SQL altogether. However, some employeers may want to know you have direct SQL experience. After learning `dplyr`, you will be able to pick up SQL very quickly. Here is a reference to help you make the small step to direct SQL queries (https://db.rstudio.com/advanced/translation/).

# dplyr single table verbs

1. `filter()` and `slice()`: find/keep certain rows
2. `arrange()`: reorder rows
3. `select()`: find/keep certain columns
   - `rename()` will change the column name
4. `mutate()`: add/create new variables
   - `transmute()`: only return new variables

# dplyr single table verbs

All verbs work similarly:

1. The first argument is a data frame.
2. The subsequent arguments describe what to do with the data frame. You can refer to columns in the data frame directly without using `$`.
3. The result is a new data frame.

Together these properties make it easy to chain together multiple simple steps to achieve a complex result.

Again, the Data Transformation Cheatsheet is a handy reference.

# Select rows with `filter()` and `slice()`

# Select rows by position with slice()

To select rows by position, use slice():

```
slice(flights, 5:8)        # selects the 5th - 8th row
#> # A tibble: 4 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      554            600        -6      812
#> 2  2013     1     1      554            558        -4      740
#> 3  2013     1     1      555            600        -5      913
#> 4  2013     1     1      557            600        -3      709
#> # ... with 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>
```

# Select rows by values with `filter()`

`filter()` allows you to subset observations according to specific criteria.

- The first argument is the name of the data frame.
- The second and subsequent arguments are the expressions that filter the data frame (think **and**).
- For example, we can select all flights on January 1st with:

```r
filter(flights, month == 1, day == 1)
```

**Your Turn #1**

1. Find all flights with a destination (`dest`) of Chicago O’Hare (‘ORD’).
2. Find all United (‘UA’) flights with a destination (`dest`) of Chicago O’Hare (‘ORD’).

# Relational Operators for Numeric Vectors

R provides the standard suite of *numeric* comparison operators: $>$, $>=$, $<$, $<=$, $!=$ (not equal), and $==$ (equal).

**Your Turn #2**

1. Find all flights that departed (`dep_time`) after 8pm (20:00).
2. Find all United ('UA') flights that departed (`dep_time`) after 8pm (20:00),
   with a destination (`dest`) of Chicago O'Hare ('ORD').

# One equals or two?

When you’re starting out with R, the easiest mistake to make is to use = instead of == when testing for equality. When this happens you’ll get an error message with a hint:

```
filter(flights, month = 1)
#> `month` (`month = 1`) must not be named, do you need `==`?
```

Whenever you see this message, check for = instead of ==.

# Relational Operators for Character Vectors (and Factors)

For *categorical* vectors:

- == equal to
- != not equal to
- `%in%` element of set (use: `x %in% set`)

```
x = c("aa", "bb", "aa", "bb", "aa", "cc", "dd")
x == "aa"
#> [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE
x != "aa"
#> [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE  TRUE
x %in% c("aa","bb")
#> [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE
!(x %in% c("aa","bb"))  # x not in set
#> [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
```

# Logical Operators

Multiple arguments to `filter()` are combined with “and”.

```
#- select flights with dest of BHM *and* December
filter(flights, dest=="BHM", month == 12)
```

To get more complicated expressions, you can use Boolean operators. The `|` is read as “or”

```
#- select flights with Nov *or* Dec
filter(flights, month == 11 | month == 12)
```

```
#- dest of BHM *and* (Nov *or* Dec)
filter(flights, dest=="BHM", month == 11 | month == 12)
```

# Logical Dangers

**Your Turn #3**

Find all flights with destination of `DCA` *or* `IAD`.

# Logical Dangers

Beware of a common mistake:

```r
filter(flights, month == 11 | 12)
```

Note the order isn't like English. This expression doesn't find on months that equal 11 or 12. Instead it finds all months that equal `11 | 12`, which is `TRUE`:

```r
11 | 12
#> [1] TRUE
```

In a numeric context (like here), `TRUE` is interpreted as a `1`, so this finds all flights in *January*, not November or December.

# Values in a set

Instead of many *OR* statements, you can use the helpful `%in%` shortcut:

```
filter(flights, month %in% c(11, 12))
```

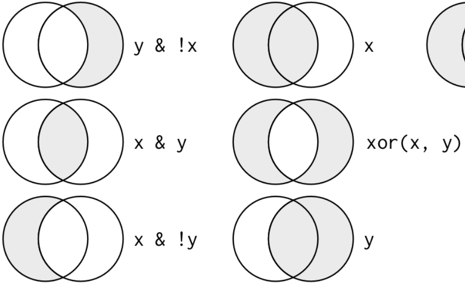
Or `between()`

```
filter(flights, between(month, 11, 12))
```

The function `between(x, left, right)` is a shortcut for `x >= left & x<= right` (inclusive).

# More Logical and Relational Operators

- I have compiled a list of some common logical and relational operators
- Complete set of Boolean operations from the R for Data Science book:



## Your Turn #4 : filter()

Find all the flights that:

a. Departed in July
b. That flew to Houston (`IAH` or `HOU`)
c. Departed in July and flew to Houston
d. Flew to `Hou` or Originated from 'JFK'
e. That were delayed by more than two hours
f. That arrived more than two hours late, but didn't leave late
g. Had an arrival time earlier than departure time

Understand how each variable is coded (e.g. the integer 1 = January, the integer 517 = 5:17am, etc.).

# Solutions

# Arranging (ordering) rows with `arrange()`

# Arrange rows with `arrange()`

- `arrange()` works similarly to `filter()` except that instead of filtering or selecting rows, it reorders them.
- It takes a data frame, and a set of column names (or more complicated expressions) to order by.
- If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns.
- Order by `year`, then `month`, then `day`:

```
arrange(flights, year, month, 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     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
#> 6  2013     1     1      554            558        -4      740
#> # ... with 3.368e+05 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>
```

# Descending Order

- By default, `arrange()` orders from smallest to largest
- Use `desc()` to order a column in descending order:

```
arrange(flights, desc(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    10    30     2400           2359         1      327
#> 2  2013    11    27     2400           2359         1      515
#> 3  2013    12     5     2400           2359         1      427
#> 4  2013    12     9     2400           2359         1      432
#> 5  2013    12     9     2400           2250        70       59
#> 6  2013    12    13     2400           2359         1      432
#> # ... with 3.368e+05 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>
```

- This works on categorical data too (alphabetical order)
- This works on factors too (ordered by `levels`)

### Your Turn #5 : arrange()

a. Sort flights to find the most delayed flights
b. Sort flights to find the least delayed flights
c. Sort flights by destination and break ties by arrival delay
d. Sort flights to find highest average flight speed (`distance/air_time`)

# Solutions

# Select columns with `select()`

# Select columns with `select()`

- It's not uncommon to get datasets with hundreds or even thousands of variables.
- In this case, the first challenge is often narrowing in on the variables you're actually interested in.
- `select()` allows you to rapidly zoom in on a useful subset using operations based on the names or positions of the variables.
- Select columns **by name**

```
select(flights, year, month, day)  # keep year, month, and day columns
```

- Select columns **by position**

```
select(flights, 1:3)  # keep first 3 columns
```

# Other ways to select columns

- *De*select or drop columns using the − (minus) symbol

```r
select(flights, -year, -month, -day)  # keep all except year, month, day
```

```r
select(flights, -(1:3))  # keep all except first 3 columns
```

- Select range of columns by name

```r
# Select all columns between year and day (inclusive)
select(flights, year:day)
# Select all columns except those from year to day (inclusive)
select(flights, -(year:day))
```

# Yet more ways to select columns

There are a number of helper functions you can use within `select()`:

- `starts_with("abc")`: matches names that begin with “abc”.
- `ends_with("xyz")`: matches names that end with “xyz”.
- `contains("ijk")`: matches name that contain “ijk”.
- `matches("(.)\\1")`: selects variables that match a regular expression.
  This one matches any variables that contain repeated characters.
  You’ll learn more about regular expressions later in the course
- `num_range("x", 1:3)` matches `x1`, `x2` and `x3`.
- `one_of(x)` selects any names in the vector `x`

See `?select` and Data Transformation Cheatsheet for more details.

# Related functionality: `rename()`

## Use `rename()` function to rename a column

```
rename(flights, tail_number = tailnum)
#> # 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
#> 6  2013     1     1      554            558        -4      740
#> # ... with 3.368e+05 more rows, and 12 more variables:
#> #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tail_number <chr>, origin <chr>, dest <chr>, air_time <dbl>,
#> #   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

- Note: this returns a full data frame. It does *not* modify the original.
- To apply the renaming, use `flights = rename(flights, tail_number = tailnum)`

# Re-arrange Columns

- The column order can be rearranged with `select()`. This is especially helpful for viewing on the screen/console

```
select(flights, distance, air_time, origin, dest, carrier)
#> # A tibble: 336,776 x 5
#>   distance air_time origin dest  carrier
#>      <dbl>    <dbl> <chr>  <chr> <chr>
#> 1     1400      227 EWR    IAH   UA
#> 2     1416      227 LGA    IAH   UA
#> 3     1089      160 JFK    MIA   AA
#> 4     1576      183 JFK    BQN   B6
#> 5      762      116 LGA    ATL   DL
#> 6      719      150 EWR    ORD   UA
#> # ... with 3.368e+05 more rows
```

# Renaming Columns with `select()`

The `select()` function also allows renaming on the fly

```
select(flights, dist=distance,
       `what time is it?`=air_time,
       new_name=carrier)
#> # A tibble: 336,776 x 3
#>    dist `what time is it?` new_name
#>   <dbl>              <dbl> <chr>
#> 1  1400                227 UA
#> 2  1416                227 UA
#> 3  1089                160 AA
#> 4  1576                183 B6
#> 5   762                116 DL
#> 6   719                150 UA
#> # ... with 3.368e+05 more rows
```

Hint: If you really, really want to use spaces or strange characters in column names, use *back-ticks* (shown above)

# Add or modify variables with `mutate()`

# Add or modify variables with `mutate()`

- The job of `mutate()` is to add new (or modify) columns that are functions of existing columns.
- `mutate()` always adds the new columns at the end of the data frame in order created

```
flights_sml <- select(flights,         # reduce variables
  year:day,
  ends_with("delay"),
  distance,
  air_time
)

mutate(flights_sml,
  gain = arr_delay - dep_delay,        # add gain variable
  speed = distance / (air_time / 60)   # add speed variable (in mph)
)
#> # A tibble: 336,776 x 9
#>    year month   day dep_delay arr_delay distance air_time  gain speed
#>   <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl> <dbl> <dbl>
#> 1  2013     1     1         2        11     1400      227     9  370.
#> 2  2013     1     1         4        20     1416      227    16  374.
#> 3  2013     1     1         2        33     1089      160    31  408.
#> 4  2013     1     1        -1       -18     1576      183   -17  517.
#> 5  2013     1     1        -6       -25      762      116   -19  394.
#> 6  2013     1     1        -4        12      719      150    16  288.
#> # ... with 3.368e+05 more rows
```

# mutate() function

- Note that you can refer to columns that you've just created:

```
mutate(flights_sml,
  gain = arr_delay - dep_delay,
  hours = air_time / 60,
  gain_per_hour = gain / hours       # used the newly created variables
)
#> # A tibble: 336,776 x 10
#>    year month   day dep_delay arr_delay distance air_time  gain hours
#>   <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl> <dbl> <dbl>
#> 1  2013     1     1         2        11     1400      227     9  3.78
#> 2  2013     1     1         4        20     1416      227    16  3.78
#> 3  2013     1     1         2        33     1089      160    31  2.67
#> 4  2013     1     1        -1       -18     1576      183   -17  3.05
#> 5  2013     1     1        -6       -25      762      116   -19  1.93
#> 6  2013     1     1        -4        12      719      150    16  2.5
#> # ... with 3.368e+05 more rows, and 1 more variable: gain_per_hour <dbl>
```

`mutate()` is also used to modify the columns (e.g. `recode()` or change data type). E.g., `mutate(flights, flight = as.character(flight)` will change `flight` column to a character.

# transmute() to only keep new columns

If you only want to keep the newly created columns, use transmute() instead of mutate() + select()

```
transmute(flights,
  gain = arr_delay - dep_delay,
  hours = air_time / 60,
  gain_per_hour = gain / hours
)
#> # A tibble: 336,776 x 3
#>    gain hours gain_per_hour
#>   <dbl> <dbl>         <dbl>
#> 1     9  3.78          2.38
#> 2    16  3.78          4.23
#> 3    31  2.67         11.6
#> 4   -17  3.05         -5.57
#> 5   -19  1.93         -9.83
#> 6    16  2.5           6.4
#> # ... with 3.368e+05 more rows
```

# Using aggregate functions in `mutate()`

- For statistical analysis, we often want to compare individual values to aggregates
- E.g., create the *Z* score for the `distance` column

```
transmute(flights,
          Zdist = (distance - mean(distance))/sd(distance))
#> # A tibble: 336,776 x 1
#>      Zdist
#>      <dbl>
#> 1   0.491
#> 2   0.513
#> 3   0.0669
#> 4   0.731
#> 5 -0.379
#> 6 -0.438
#> # ... with 3.368e+05 more rows
```

For each element in the `distance` column, it subtracts the column mean and divides by the column standard deviation.

## Your Turn #6 : mutate()

a. Create a new data frame that contains only the flights that were less than 1000 miles (`distance`). Keep only the columns: `dep_delay`, `arr_delay`, `origin`, `dest`, `air_time`, and `distance`.
b. Add the $Z$-score for departure delays to the new data frame
c. Convert the departure and arrival delays into hours
d. Return only the average flight speed (in mph)
e. Calculate the mean speed

# Solutions

# Other `dplyr` functions

# Honorable Mentions: Data frame functions

- `distinct()`: retain unique/distinct rows
- `sample_n()` and `sample_frac()`: randomly sample rows
- `top_n()` / `top_frac()`: selects and orders the top n rows according to `wt`
- `add_column()` add new column in particular position
- `add_row()` adds new row(s) to the table

# Honorable Mentions: Dealing with NA's (missing values)

Dealing with missing values (NA) is important, but tedious. These can help

- `na_if(x, y)` converts the y valued elements in x to NA

```
x = c(1, 2, -99, 5, 5, -99)
na_if(x, -99)                  # replace -99 with NA
#> [1]  1  2 NA  5  5 NA
```

- `coalesce(x, y)` replaces the NA in x with y

```
x = c(1, 2, NA, 5, 5, NA)
coalesce(x, 0)               # replace NA with 0
#> [1] 1 2 0 5 5 0
```

These two functions can be used in `mutate()` to modify columns.