Introduction to R

Presented by:

BROWN
School of Public Health

Intro to R Programming for Biostatistics

Day 2 - Cleaning and Transforming Data in R

Adam J Sullivan

Piping or Chaining Data

Piping or Chaining

- We will discuss a concept that will help us greatly when it comes to working with our data.
- The usual way to perform multiple operations in one line is by nesting.

Piping or Chaining

To consider an example we will look at the data provided in the gapminder package:

```
library(gapminder)
head(gapminder)
```

```
## # A tibble: 6 × 6
##       country continent  year lifeExp      pop gdpPercap
##        <fctr>    <fctr> <int>   <dbl>    <int>     <dbl>
## 1 Afghanistan      Asia  1952  28.801  8425333  779.4453
## 2 Afghanistan      Asia  1957  30.332  9240934  820.8530
## 3 Afghanistan      Asia  1962  31.997 10267083  853.1007
## 4 Afghanistan      Asia  1967  34.020 11537966  836.1971
## 5 Afghanistan      Asia  1972  36.088 13079460  739.9811
## 6 Afghanistan      Asia  1977  38.438 14880372  786.1134
```

Nesting vs Chaining

- Let's say that we want to have the GDP per capita and life expectancy Kenya.
- Traditionally speaking we could do this in a nested manner:

```
filter(select(gapminder, country, lifeExp, gdpPercap), country=="Kenya")
```

Nesting vs Chaining

- It is not easy to see exactly what this code was doing but we can write this in a manner that follows our logic much better.
- The code below represents how to do this with chaining.

```
gapminder %>%
    select(country, lifeExp, gdpPercap) %>%
    filter(country=="Kenya")
```

Breaking Down the Code

- We now have something that is much clearer to read.
- Here is what our chaining command says:

1. Take the `gapminder` data
2. Select the variables: `country`, `lifeExp` and `gdpPercap`.
3. Only keep information from Kenya.

- The nested code says the same thing but it is hard to see what is going on if you have not been coding for very long.

Breaking Down the Code

- The result of this search is below:

```
## # A tibble: 12 × 3
##    country lifeExp gdpPercap
##     <fctr>   <dbl>     <dbl>
## 1    Kenya  42.270  853.5409
## 2    Kenya  44.686  944.4383
## 3    Kenya  47.949  896.9664
## 4    Kenya  50.654 1056.7365
## 5    Kenya  53.559 1222.3600
## 6    Kenya  56.155 1267.6132
## 7    Kenya  58.766 1348.2258
## 8    Kenya  59.339 1361.9369
## 9    Kenya  59.285 1341.9217
## 10   Kenya  54.407 1360.4850
## 11   Kenya  50.992 1287.5147
## 12   Kenya  54.110 1463.2493
```

What is %>%

- In the previous code we saw that we used `%>%` in the command you can think of this as saying ***then***.
- For example:

```
gapminder %>%
    select(country, lifeExp, gdpPercap) %>%
    filter(country=="Kenya")
```

What Does this Mean?

- This translates to:
  - Take Gapminder ***then*** select these columns select(country, lifeExp, gdpPercap) ***then*** filter out so we only keep Kenya

Why Chain?

- We still might ask why we would want to do this.
- Chaining increases readability significantly when there are many commands.
- With many packages we can replace the need to perform nested arguments.
- The chaining operator is automatically imported from the magrittr (https://github.com/smbache/magrittr) package.

User Defined Function

- Let's say that we wish to find the Euclidean distance between two vectors say, `x1` and `x2`.
- We could use the math formula:

$$\sqrt{\text{sum}(\text{x1} - \text{x2})^2}$$

User Defined Function

- In the nested manner this would be:

```
x1 <- 1:5; x2 <- 2:6
sqrt(sum((x1-x2)^2))
```

User Defined Function

- However, if we chain this we can see how we would perform this mathematically.

```
# chaining method
(x1-x2)^2 %>% sum() %>% sqrt()
```

- If we did it by hand we would perform elementwise subtraction of `x2` from `x1` ***then*** we would sum those elementwise values ***then*** we would take the square root of the sum.

User Defined Function

```
# chaining method
(x1-x2)^2 %>% sum() %>% sqrt()
```

```
## [1] 2.236068
```

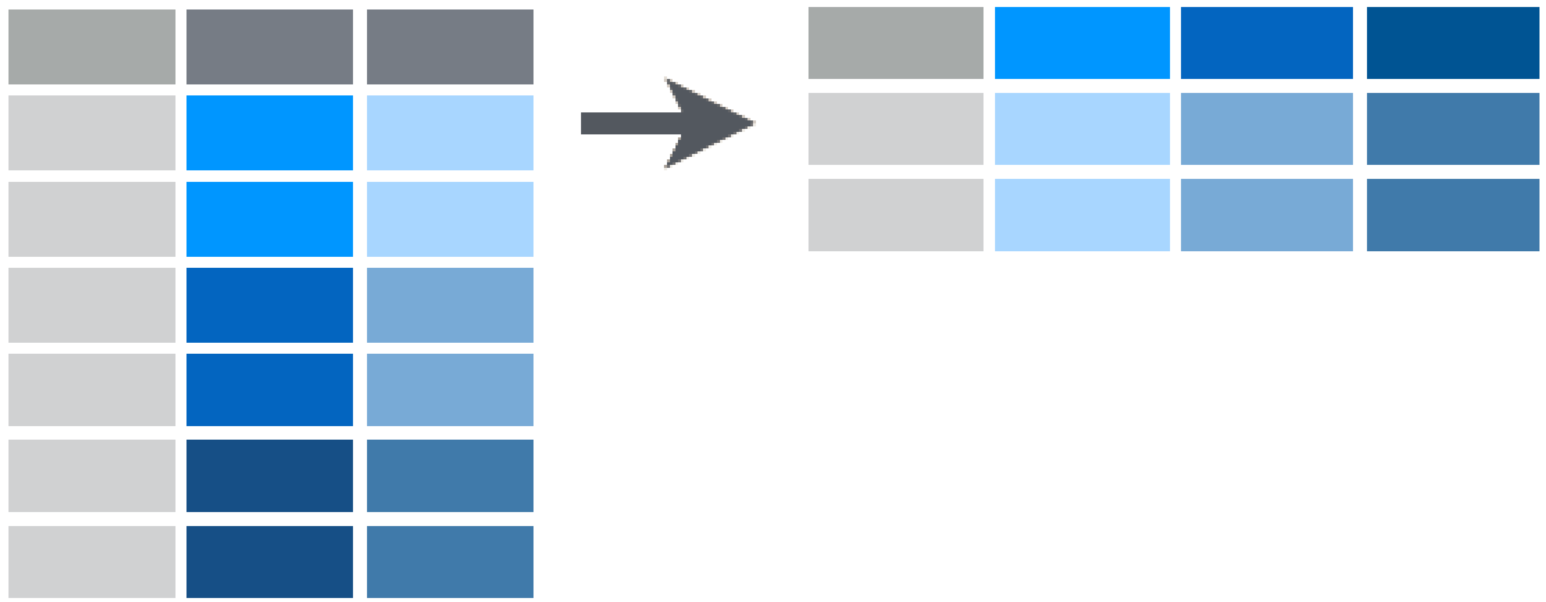
- Many of us have been performing calculations by this type of method for years, so that chaining really is more natural for us.

The `spread()` Function

The `spread()` Function

- The first `tidyr` function we will look into is the `spread()` function.
- With `spread()` it does similar to what you would expect.
- We have a data frame where some of the rows contain information that is really a variable name.
- This means the columns are a combination of variable names as well as some data.

The picture below displays this:



We can consider the following data which is table 2:

```
## # A tibble: 12 × 4
##        country  year        key      value
##         <fctr> <int>     <fctr>      <int>
## 1  Afghanistan  1999      cases        745
## 2  Afghanistan  1999 population   19987071
## 3  Afghanistan  2000      cases       2666
## 4  Afghanistan  2000 population   20595360
## 5       Brazil  1999      cases      37737
## 6       Brazil  1999 population  172006362
## 7       Brazil  2000      cases      80488
## 8       Brazil  2000 population  174504898
## 9        China  1999      cases     212258
## 10       China  1999 population 1272915272
## 11       China  2000      cases     213766
## 12       China  2000 population 1280428583
```

Notice that in the column of `key`, instead of there being values we see the following variable names:

- cases
- population

In order to use this data we need to have it so the data frame looks like this instead:

```
## # A tibble: 6 × 4
##       country  year  cases population
## *      <fctr> <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3      Brazil  1999  37737  172006362
## 4      Brazil  2000  80488  174504898
## 5       China  1999 212258 1272915272
## 6       China  2000 213766 1280428583
```

- Now we can see that we have all the columns representing the variables we are interested in and each of the rows is now a complete observation.
- In order to do this we need to learn about the `spread()` function:

The `spread()` Function

```
spread(data, key, value)
```

Where

- `data` is your dataframe of interest.
- `key` is the column whose values will become variable names.
- `value` is the column where values will fill in under the new variables created from `key`.

Piping

If we consider **piping**, we can write this as:

```
data %>%
  spread(key, value)
```

spread() Example

Now if we consider table2 , we can see that we have:

```
## # A tibble: 12 × 4
##         country  year        key      value
##          <fctr> <int>     <fctr>      <int>
## 1  Afghanistan  1999      cases        745
## 2  Afghanistan  1999 population   19987071
## 3  Afghanistan  2000      cases       2666
## 4  Afghanistan  2000 population   20595360
## 5       Brazil  1999      cases      37737
## 6       Brazil  1999 population  172006362
## 7       Brazil  2000      cases      80488
## 8       Brazil  2000 population  174504898
## 9        China  1999      cases     212258
## 10       China  1999 population 1272915272
## 11       China  2000      cases     213766
## 12       China  2000 population 1280428583
```

`spread()` Example

- Now this table was made for this example so key is the `key` in our `spread()` function and value is the `value` in our `spread()` function.
- We can fix this with the following code:

`spread()` Example

```
table2 %>%
    spread(key,value)
```

```
## # A tibble: 6 × 4
##       country  year  cases population
## *      <fctr> <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3      Brazil  1999  37737  172006362
## 4      Brazil  2000  80488  174504898
## 5       China  1999 212258 1272915272
## 6       China  2000 213766 1280428583
```

`spread()` Example

- We can now see that we have a variable named `cases` and a variable named `population`.
- This is much more tidy.

On Your Own: RStudio Practice

- We first will load tidyverse.
- If you have not installed it run the following code:
  - `install.packages("tidyverse")`
- Then load this package:
  - `library(tidyverse)`

On Your Own: RStudio Practice

- In this example we will use the dataset `population` that is part of tidyverse.
- Print this data:

```
## # A tibble: 1 × 3
##       country  year population
##         <chr> <int>      <int>
## 1 Afghanistan  1995   17586073
```

On Your Own: RStudio Practice

- You should see the table that we have above, now We have a variable named `year`, assume that we wish to actually have each year as its own variable.
- Using the `spread()` function, redo this data so that each year is a variable.
- Your data will look like this at the end:

On Your Own: RStudio Practice

```
## # A tibble: 219 × 20
##               country   `1995`   `1996`   `1997`   `1998`   `1999`
## *               <chr>    <int>    <int>    <int>    <int>    <int>
## 1         Afghanistan 17586073 18415307 19021226 19496836 19987071
## 2             Albania  3357858  3341043  3331317  3325456  3317941
## 3             Algeria 29315463 29845208 30345466 30820435 31276295
## 4      American Samoa    52874    53926    54942    55899    56768
## 5             Andorra    63854    64274    64090    63799    64084
## 6              Angola 12104952 12451945 12791388 13137542 13510616
## 7            Anguilla     9807    10063    10305    10545    10797
## 8  Antigua and Barbuda    68349    70245    72232    74206    76041
## 9           Argentina 34833168 35264070 35690778 36109342 36514558
## 10            Armenia  3223173  3173425  3137652  3112958  3093820
## # ... with 209 more rows, and 14 more variables: `2000` <int>,
## #   `2001` <int>, `2002` <int>, `2003` <int>, `2004` <int>, `2005` <int>,
## #   `2006` <int>, `2007` <int>, `2008` <int>, `2009` <int>, `2010` <int>,
## #   `2011` <int>, `2012` <int>, `2013` <int>
```

The `gather()` Function

The `gather()` Function

- The second `tidyr` function we will look into is the `gather()` function.
- With `gather()` it may not be clear what exactly is going on, but in this case we actually have a lot of column names the represent what we would like to have as data values.

The `gather()` Function Example

- For example, in the last `spread()` practice you created a data frame where variable names were individual years.
- This may not be what you want to have so you can use the gather function.

Consider `table4`:

```
## # A tibble: 3 × 3
##       country `1999` `2000`
##        <fctr>  <int>  <int>
## 1 Afghanistan    745   2666
## 2      Brazil  37737  80488
## 3       China 212258 213766
```

Table 4

- This looks similar to the table you created in the `spread()` practice.
- We now wish to change this data frame so that `year` is a variable and 1999 and 2000 become values instead of variables.

In Comes the `gather()` Function

- We will accomplish this with the gather function:

```
gather(data, key, value, ...)
```

- where
    - `data` is the data frame you are working with.
    - `key` is the name of the `key` column to create.
    - `value` is the name of the `value` column to create.
    - `...` is a way to specify what columns to gather from.

`gather()` Example

In our example here we would do the following:

```
table4 %>%
    gather("year", "cases", 2:3)
```

```
## # A tibble: 6 × 3
##       country  year  cases
##        <fctr> <chr>  <int>
## 1 Afghanistan  1999    745
## 2      Brazil  1999  37737
## 3       China  1999 212258
## 4 Afghanistan  2000   2666
## 5      Brazil  2000  80488
## 6       China  2000 213766
```

- You can see that we have created 2 new columns called `year` and `cases`.
- We filled these with the previous 2nd and 3rd columns.
- Note that we could have done this in many different ways too.

- For example if we knew the years but not which columns we could do this:

```
table4 %>%
    gather("year", "cases", "1999":"2000")
```

- We could also see that we want to gather all columns except the first so we could have used:

```
table4 %>%
    gather("year", "cases", -1)
```

On Your Own: RStudio Practice

- Create `population2` from last example:

```
population 2 <- population %>%
                    spread(year, population)
```

- Now gather the columns that are labeled by year and create columns `year` and `population`.

On Your Own: RStudio Practice

In the end your data frame should look like:

```
## # A tibble: 2 × 3
##       country  year population
##         <chr> <int>      <int>
## 1 Afghanistan  1995   17586073
## 2 Afghanistan  1996   18415307
```

The `dplyr` Package

The `dplyr` Package

- Now that we have started to tidy up our data we can see that we have a need to transform this data.
- We may wish to add additional variables.
- The `dplyr` package allows us to further work with our data.

`dplyr` Functionality

- With `dplyr` we have five basic verbs that we will learn to work with:

  - `filter()`
  - `select()`
  - `arrange()`
  - `mutate()`
  - `summarize()`

`dplyr` Functionality

- We also will consider:

```
- joins

- group_by()
```

`nycflights13` Data

- For the purposes of this example we will consider looking at the package `nycflights13`.
- This is a dataset that has all flights in and out of NYC in 2013.
- We also will be using the `dyplr` package from `tidyverse`:

```
library(dplyr)
library(nycflights13)
```

Filtering

Filtering

- At this point we will consider how we pick the rows of the data that we wish to work with.
- If you consider many modern data sets, we have so much information that we may not want to bring it all in at once.
- R brings data into the RAM of your computer. This means you can be limited for what size data you can bring in at once.
- Very rarely do you need the entire data set.
- We will focus on how to pick the rows or observations we want now.

Enter the `filter()` Function

- The `filter()` function chooses rows that meet a specific criteria.
- We can do this with Base R functions or with `dplyr`.

Filtering Example

- Let's say that we want to look at the flights data but we are only interested in the data from the first day of the year.
- We could do this without learning a new command and use indexing which we learned yesterday.

```
flights[flights$month==1 & flights$day==1, ]
```

```
## # A tibble: 842 × 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
## 7   2013     1     1      555            600        -5      913
## 8   2013     1     1      557            600        -3      709
## 9   2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## # ... with 832 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 Example

- Now this is not very difficult to do, what we have is that we are working with `flights` and we only want to keep the rows of data there `month==1` and `day==1`.
- However we could use the `filter()` function to do this in a much easier to read format:

`filter()` Function

```
filter(.data, ...)
```

where

- `.data` is a tibble.
- `...` is a set of arguments the data you want returned needs to meet.

Filtering Example

- This means in our example we could perform the following:

```
flights %>%
    filter(month==1, day==1)
```

Finally we could also only do one filtering at a time and chain it:

```
flights %>%
    filter(month==1) %>%
    filter(day==1)
```

Further Filtering

- `filter()` supports the use of multiple conditions where we can use Boolean.
- For example if we wanted to consider only flights that depart between 0600 and 0605 we could do the following:

```
flights %>% filter(dep_time >= 600, dep_time <= 605)
```

Further Filtering

```
## # A tibble: 2,460 × 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
## 1   2013     1     1      600            600         0      851
## 2   2013     1     1      600            600         0      837
## 3   2013     1     1      601            600         1      844
## 4   2013     1     1      602            610        -8      812
## 5   2013     1     1      602            605        -3      821
## 6   2013     1     2      600            600         0      814
## 7   2013     1     2      600            605        -5      751
## 8   2013     1     2      600            600         0      819
## 9   2013     1     2      600            600         0      846
## 10  2013     1     2      600            600         0      737
## # ... with 2,450 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>
```

Further Filtering

- We can also use the `filter()` function to remove missing data for us.
- Previously we learned about a class of functions called `is.`*foo*`()` where *foo* represents a data type.
- We could choose to only use flights that have a departure time.
- That means we wish to not have missing data for departure time:

```
flights %>% filter(!is.na(dep_time))
```

On Your Own: RStudio Practice

Using the `filter()` function and chaining:

- Choose only rows associated with
  - United Airlines (UA)
  - American Airlines (AA)

On Your Own: RStudio Practice

Your end result should be:

```
## # A tibble: 91,394 × 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      554            558        -4      740
## 5   2013     1     1      558            600        -2      753
## 6   2013     1     1      558            600        -2      924
## 7   2013     1     1      558            600        -2      923
## 8   2013     1     1      559            600        -1      941
## 9   2013     1     1      559            600        -1      854
## 10  2013     1     1      606            610        -4      858
## # ... with 91,384 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>
```

Selecting

Selecting

- The next logical step would be to select the columns we want as well.
- Many times we have so many columns that we are no interested in for a particular analysis. - Instead of slowing down your analysis by continuing to run through extra data, we could just select the columns we care about.

Enter the `select()` Function

- The `select()` function chooses columns that we specify.
- Again we can do this with base functions or with `dplyr`.
- We feel that as you continue on with your R usage that you will most likely want to go the route of `dplyr` functions instead.

Select Example

- Let's say that we want to look at the flights data but we are really only interested in the arrival time, departure time and the particular flight number.
- This seems reasonable if we are a customer and wanted to only know these pieces of information. We could do this with indexing :

```
flights[, c("dep_time", "arr_time", "flight")]
```

`select()` Function

```
select(.data, ...)
```

where

- `.data` is a tibble.
- `...` are the columns that you wish to have in bare (no quotations)

Selecting Example Continued

We could then do the following

```
flights %>%
  filter(dep_time, arr_time, flight)
```

Selecting Example Continued

```
## # A tibble: 328,063 × 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
## 7   2013     1     1      555            600        -5      913
## 8   2013     1     1      557            600        -3      709
## 9   2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## # ... with 328,053 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>
```

Removing Columns

- We may wish to pick certain columns that we wish to have but we also may want to remove certain columns.
- It is quite common to de-identify a dataset before actually distributing it to a research team. - The `select()` function will also remove columns.

Removing Columns

- Lets say that we wished to remove the `month` and `day` of the flights:

```
flights %>%
  select(-month,-day)
```

Removing Columns

We also could use a vector for this:

```
cols <- c("month", "day")
flights %>%
  select(-one_of(cols))
```

Removing Columns

- We can also remove columns that contain a certain phrase in the name.
- If we were interested in removing any columns that had to do with time we could search for the word "time" in the data and remove them:

```
flights %>%
  select(-contains("time"))
```

Removing Columns

```
## # A tibble: 336,776 × 13
##     year month   day dep_delay arr_delay carrier flight tailnum origin
##    <int> <int> <int>     <dbl>     <dbl>   <chr>  <int>   <chr>  <chr>
## 1   2013     1     1         2        11      UA   1545  N14228    EWR
## 2   2013     1     1         4        20      UA   1714  N24211    LGA
## 3   2013     1     1         2        33      AA   1141  N619AA    JFK
## 4   2013     1     1        -1       -18      B6    725  N804JB    JFK
## 5   2013     1     1        -6       -25      DL    461  N668DN    LGA
## 6   2013     1     1        -4        12      UA   1696  N39463    EWR
## 7   2013     1     1        -5        19      B6    507  N516JB    EWR
## 8   2013     1     1        -3       -14      EV   5708  N829AS    LGA
## 9   2013     1     1        -3        -8      B6     79  N593JB    JFK
## 10  2013     1     1        -2         8      AA    301  N3ALAA    LGA
## # ... with 336,766 more rows, and 4 more variables: dest <chr>,
## #   distance <dbl>, hour <dbl>, minute <dbl>
```

Unique Observations

- Many times we have a lot of repeats in our data.
- If we just would like to have an account of all things included then we can use the `unique()` command.
- Lets assume that we wish to know the origin of a flight and its destination.
- We do not want to have every flight listed over and over again so we ask for unique values:

```
flights %>%
  select(origin, dest) %>%
  unique()
```

On Your Own: RStudio Practice

- Consider the flights data: `flights`.
  1. Select all but the `year` column.
  2. Remove the month and day from them.
  3. Select values which contain "time" in them.
  4. Chain these together so that you run a command and it does all of these things.

On Your Own: RStudio Practice

Your answer should look like:

```
## # A tibble: 336,776 × 6
##    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
## 6       554            558      740            728      150
## 7       555            600      913            854      158
## 8       557            600      709            723       53
## 9       557            600      838            846      140
## 10      558            600      753            745      138
## # ... with 336,766 more rows, and 1 more variables: time_hour <dttm>
```