DATA SOCIETY®

Intro to R programming - day 4

"One should look for what is and not what he thinks should be."
-Albert Einstein.

Module completion checklist

Objective	Complete
Rank data using the arrange function	
Select specific variables, sometimes using specific rules, using the select command	
Derive new variables from the existing variables using the mutate and transmute commands	
Perform multiple functions with the pipe operator (%>%)	
Summarize columns using the summary and group by functions	
Convert wide to long data using tidyr package	
Manipulate columns by using the separate and unite functions	

Directory settings

- First, let's make sure to set our directories correctly, this way, we will not have to worry about this throughout the course

```
# Set `main_dir` to the location of your `af-werx` folder (for Mac/Linux).
main_dir = "~/Desktop/af-werx"

# Set `main_dir` to the location of your `af-werx` folder (for Windows).
main_dir = "C:/Users/[username]/Desktop/af-werx"

# Make `data_dir` from the `main_dir` and
# remainder of the path to data directory.
data_dir = paste0(main_dir, "/data")

# Make `plots_dir` from the `main_dir` and
# remainder of the path to plots directory.
plot_dir = paste0(main_dir, "/plots")
```

Set working directory and load data

```r
setwd(data_dir)
load("tidyr_tables.RData")

# Load the dataset and save it as 'flights'.
# It is native to R so we can load it like this.
flights = nycflights13::flights
```

Installing packages

```r
# Install package.
install.packages("tidyverse")

# Load the package into the environment.
library(tidyverse)

# View package documentation.
library(help = "tidyverse")
```

Documentation for package 'tidyv... ×

```
                Information on package 'tidyverse'

Description:

Package:            tidyverse
Title:              Easily Install and Load the 'Tidyverse'
Version:            1.2.1
Authors@R:          c( person("Hadley", "Wickham", , "hadley@rstudio.com", role =
                    c("aut", "cre")), person("RStudio", role = c("cph", "fnd")) )
Description:        The 'tidyverse' is a set of packages that work in harmony because
                    they share common data representations and 'API' design. This package
                    is designed to make it easy to install and load multiple 'tidyverse'
                    packages in a single step. Learn more about the 'tidyverse' at
                    <https://tidyverse.org>.
License:            GPL-3 | file LICENSE
URL:                http://tidyverse.tidyverse.org,
                    https://github.com/tidyverse/tidyverse
BugReports:         https://github.com/tidyverse/tidyverse/issues
Imports:            broom (>= 0.4.2), cli (>= 1.0.0), crayon (>= 1.3.4), dplyr (>=
                    0.7.4), dbplyr (>= 1.1.0), forcats (>= 0.2.0), ggplot2 (>= 2.2.1),
                    haven (>= 1.1.0), hms (>= 0.3), httr (>= 1.3.1), jsonlite (>= 1.5),
                    lubridate (>= 1.7.1), magrittr (>= 1.5), modelr (>= 0.1.1), purrr (>=
                    0.2.4), readr (>= 1.1.1), readxl (>= 1.0.0), reprex (>= 0.1.1), rlang
                    (>= 0.1.4), rstudioapi (>= 0.7), rvest (>= 0.3.2), stringr (>=
                    1.2.0), tibble (>= 1.3.4), tidyr (>= 0.7.2), xml2 (>= 1.1.1)
Suggests:           feather (>= 0.3.1), knitr (>= 1.17), rmarkdown (>= 1.7.4)
VignetteBuilder:    knitr
Encoding:           UTF-8
LazyData:           true
RoxygenNote:        6.0.1
NeedsCompilation:   no
Packaged:           2017-11-14 00:57:48 UTC; hadley
Author:             Hadley Wickham [aut, cre], RStudio [cph, fnd]
Maintainer:         Hadley Wickham <hadley@rstudio.com>
Repository:         CRAN
Date/Publication:   2017-11-14 18:09:56 UTC
Built:              R 3.4.3; ; 2018-02-14 21:15:03 UTC; unix
```

Recap: basics of dplyr

- There are six functions that provide the verbs for the language of data manipulation
- They will most definitely make your life as a data scientist easier
- They are:

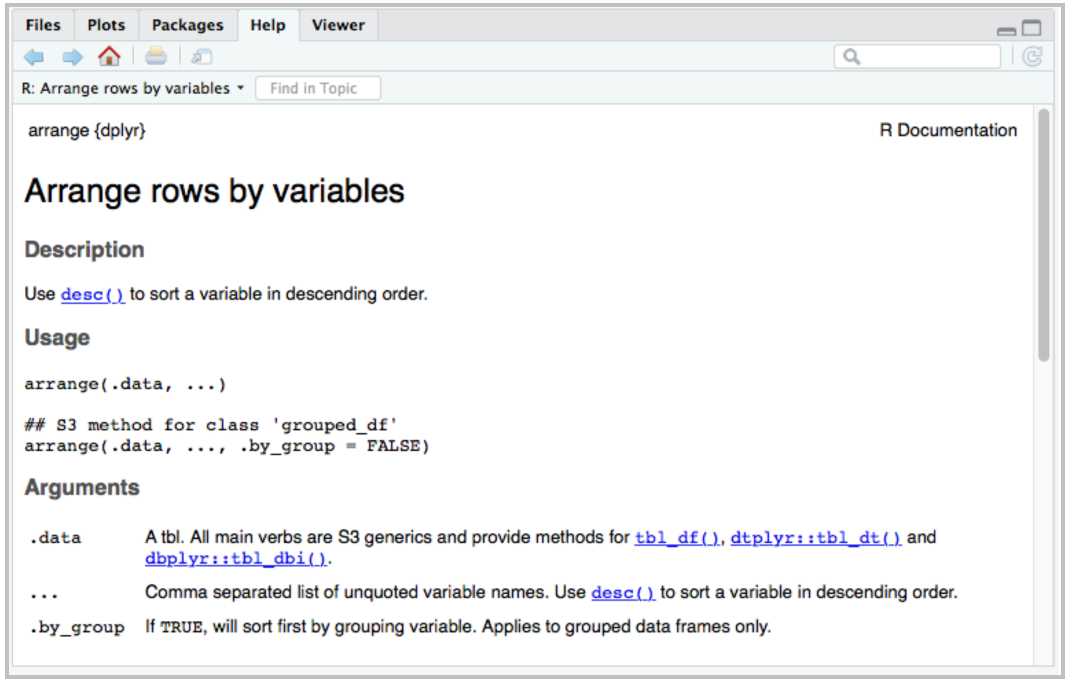
Function	Use Case	Data Type
`filter`	Pick observations by their value	All data types
`arrange`	Reorder the rows	All data types
`select`	Pick variables by their names	All data types
`mutate`	Create new variables with functions of existing variables	All data types
`summarise`	Collapse many values down to a single summary	All data types
`group_by`	Allows the first five functions to operate on a dataset group by group	All data types

Arrange

```
?dplyr::arrange

arrange(df,            #<- dataframe
        arrange_cond1, #<- column by which
                       #   to arrange
        ...)           #<- other args.
```

- `arrange` is used to change the order of rows within the specified column(s)
- `arrange` is the equivalent of `sort` in SAS or `order by` in SQL
- When using multiple columns with `arrange`, the additional columns will be used to break ties in the values of preceding columns



Files Plots Packages Help Viewer

R: Arrange rows by variables

Find in Topic

arrange {dplyr} R Documentation

Arrange rows by variables

Description

Use `desc()` to sort a variable in descending order.

Usage

```
arrange(.data, ...)

## S3 method for class 'grouped_df'
arrange(.data, ..., .by_group = FALSE)
```

Arguments

`.data`	A tbl. All main verbs are S3 generics and provide methods for `tbl_df()`, `dtplyr::tbl_dt()` and `dbplyr::tbl_dbi()`.
`...`	Comma separated list of unquoted variable names. Use `desc()` to sort a variable in descending order.
`.by_group`	If `TRUE`, will sort first by grouping variable. Applies to grouped data frames only.

Arrange example

```
# Arrange data by year, then month, and then day.
arrange(flights, #<- dataframe we want to arrange
        year,    #<- 1st: arrange by year
        month,   #<- 2nd: arrange by month
        day)     #<- 3rd: arrange by 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
 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 336,766 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>
```

Arrange options

- `arrange` by default sorts everything in ascending order; to arrange in descending, use `desc`

```
# Arrange data by year, descending month and then day.
arrange(flights,      #<- dataframe we want to arrange
        year,         #<- 1st: arrange by year
        desc(month),  #<- 2nd: arrange by month in descending order
        day)          #<- 3rd: arrange by 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    12     1       13           2359        14      446
 2  2013    12     1       17           2359        18      443
 3  2013    12     1      453            500        -7      636
 4  2013    12     1      520            515         5      749
 5  2013    12     1      536            540        -4      845
 6  2013    12     1      540            550       -10     1005
 7  2013    12     1      541            545        -4      734
 8  2013    12     1      546            545         1      826
 9  2013    12     1      549            600       -11      648
10  2013    12     1      550            600       -10      825
# … with 336,766 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>
```

- You can now see that the month at the top of the dataset is December (i.e. 12th month)

Arrange with NA values

- Missing values are **always** sorted at the end

```
# Create a dataframe with 2 columns.
NA_df = data.frame(x = c(1, NA, 2),   #<- column x with 3 entries with 1 NA
                   y = c(1, 2, 3))    #<- column y with 3 entries

# Arrange data with missing values.
arrange(NA_df, x)
```

```
   x y
1  1 1
2  2 3
3 NA 2
```

```
# Even when we use `desc`, the `NA` is taken to the last row.
arrange(NA_df, desc(x))
```

```
   x y
1  2 3
2  1 1
3 NA 2
```

Module completion checklist

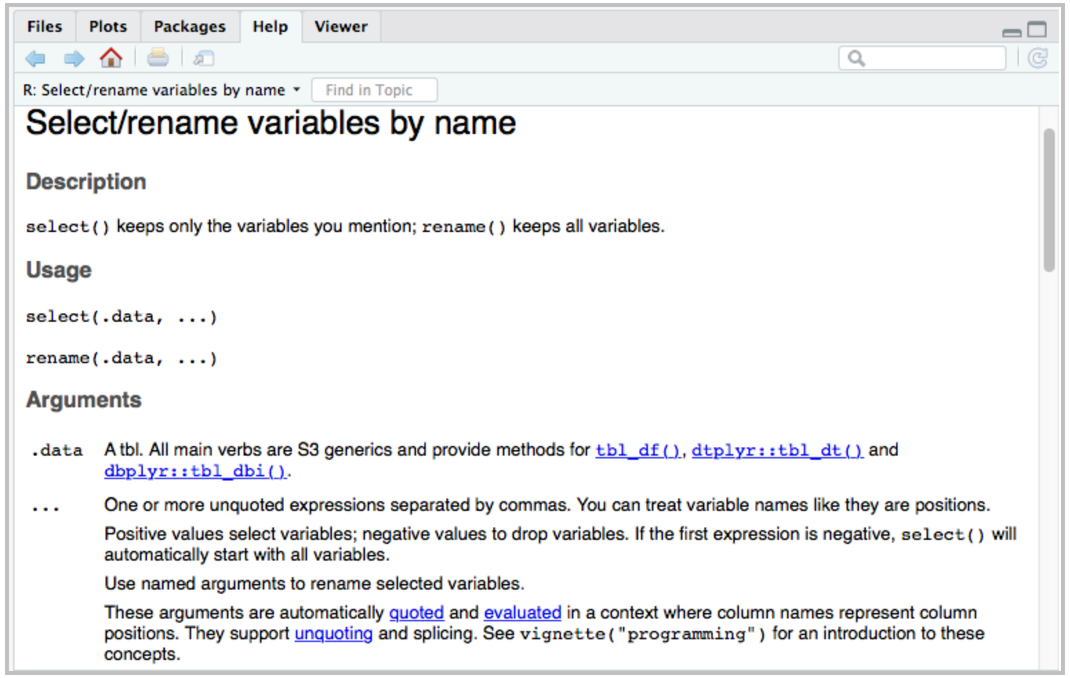
Objective	Complete
Rank data using the arrange function	✔
Select specific variables, sometimes using specific rules, using the select command	
Derive new variables from the existing variables using the mutate and transmute commands	
Perform multiple functions with the pipe operator (%>%)	
Summarize columns using the summary and group by functions	
Convert wide to long data using tidyr package	
Manipulate columns by using the separate and unite functions	

Select

```
?dplyr::select

select(df,            #<- dataframe
       select_cond1, #<- selection rule(s)
       ...)
```

- `select` helps you select specific columns within your dataframe
- We often use this function with pipes. We will cover pipes (`%>%`) later in this lesson
- The selection criteria can be written in multiple ways, shown in the next couple of slides



Select a subset

- You can simply specify each column name

```
# Select columns from `flights` dataframe.
select(flights, #<- specify the dataframe
       year,    #<- specify the 1st column
       month,   #<- specify the 2nd column
       day)     #<- specify the 3rd column
```

```
# A tibble: 336,776 x 3
    year month   day
   <int> <int> <int>
 1  2013     1     1
 2  2013     1     1
 3  2013     1     1
 4  2013     1     1
 5  2013     1     1
 6  2013     1     1
 7  2013     1     1
 8  2013     1     1
 9  2013     1     1
10  2013     1     1
# … with 336,766 more rows
```

- You can also specify a range of columns with the range operator (i.e. `:`)

```
# Select columns from `flights` dataframe.
select(flights,  #<- specify dataframe
       year:day) #<- specify range of cols
```

```
# A tibble: 336,776 x 3
    year month   day
   <int> <int> <int>
 1  2013     1     1
 2  2013     1     1
 3  2013     1     1
 4  2013     1     1
 5  2013     1     1
 6  2013     1     1
 7  2013     1     1
 8  2013     1     1
 9  2013     1     1
10  2013     1     1
# … with 336,766 more rows
```

Select by excluding

- Finally, you can select by excluding certain columns using the exclusion operator (i.e. –)

```
# Select multiple columns from `flights` dataframe
# by providing which columns to exclude in selection.
select(flights,      #<- specify the dataframe
       -(year:day))  #<- specify the range of columns to exclude
```

```
# A tibble: 336,776 x 16
   dep_time sched_dep_time dep_delay arr_time sched_arr_time arr_delay
      <int>          <int>     <dbl>    <int>          <int>     <dbl>
 1      517            515         2      830            819        11
 2      533            529         4      850            830        20
 3      542            540         2      923            850        33
 4      544            545        -1     1004           1022       -18
 5      554            600        -6      812            837       -25
 6      554            558        -4      740            728        12
 7      555            600        -5      913            854        19
 8      557            600        -3      709            723       -14
 9      557            600        -3      838            846        -8
10      558            600        -2      753            745         8
# … with 336,766 more rows, and 10 more variables: carrier <chr>,
#   flight <int>, tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
#   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Helper functions for select

- There are multiple functions you can use with `select` that act like `regex` but in a more simplified manner
- Here are some of the more commonly used helper functions:

Helper Function	Use Case
`starts_with("abc")`	matches names that begin with "abc"
`ends_with("xyz")`	matches names that end with "xyz"
`contains("ijk")`	matches names that contain "ijk"
`num_range("x", 1:3)`	matches "x1", "x2" and "x3"

Knowledge check 1

Exercise 1

Module completion checklist

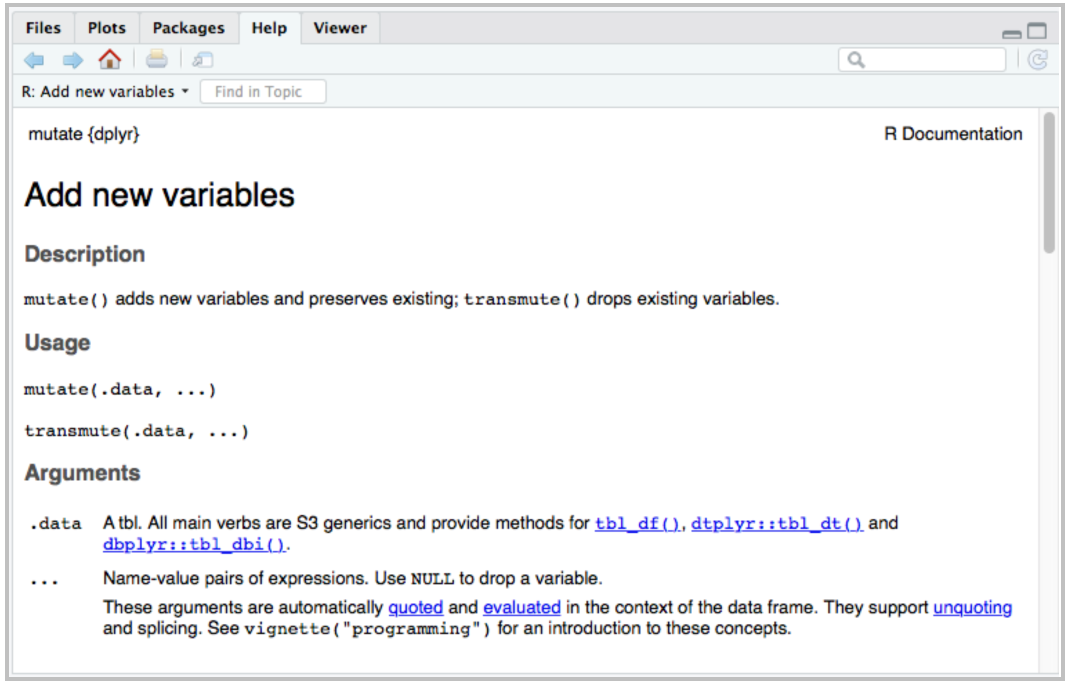
Objective	Complete
Rank data using the arrange function	✔
Select specific variables, sometimes using specific rules, using the select command	✔
Derive new variables from the existing variables using the mutate and transmute commands	
Perform multiple functions with the pipe operator (%>%)	
Summarize columns using the summary and group by functions	
Convert wide to long data using tidyr package	
Manipulate columns by using the separate and unite functions	

Mutate

```
?dplyr::mutate

mutate(df,          # <- dataframe
       new_col1, # <- rule(s) for new col
       ...)
```

- `mutate` is an essential function to dplyr, it allows us to create new variables using the current data and append these variables to the existing dataframe
- Mutate always adds columns to the end of the dataset, so we want to be able to see the last columns



Mutate

- Create the dataset using `select`

```
# Let's select columns of `flights` dataframe and save them as `flights_sml`.
flights_sml = select(flights,                 #<- specify dataframe
                     year:day,                #<- specify range of columns to include
                     ends_with("delay"),      #<- find all columns that end with `delay`
                     distance,                #<- select `distance` column
                     air_time)                #<- select `air_time` column
flights_sml
```

```
# A tibble: 336,776 x 7
    year month   day dep_delay arr_delay distance air_time
   <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl>
 1  2013     1     1         2        11     1400      227
 2  2013     1     1         4        20     1416      227
 3  2013     1     1         2        33     1089      160
 4  2013     1     1        -1       -18     1576      183
 5  2013     1     1        -6       -25      762      116
 6  2013     1     1        -4        12      719      150
 7  2013     1     1        -5        19     1065      158
 8  2013     1     1        -3       -14      229       53
 9  2013     1     1        -3        -8      944      140
10  2013     1     1        -2         8      733      138
# … with 336,766 more rows
```

Adding to the dataframe

1. The first argument is the dataframe
2. The following arguments are the columns that we would like to add to the dataframe

```
# Add two columns `gain` and `speed` to `flights_sml`.
mutate(flights_sml,                       #<- specify the dataframe
       gain = arr_delay - dep_delay,      #<- create `gain` column by subtracting departure delay
                                          #   from arrival delay
       speed = distance / air_time * 60) #<- create `speed` from distance and air time columns
```

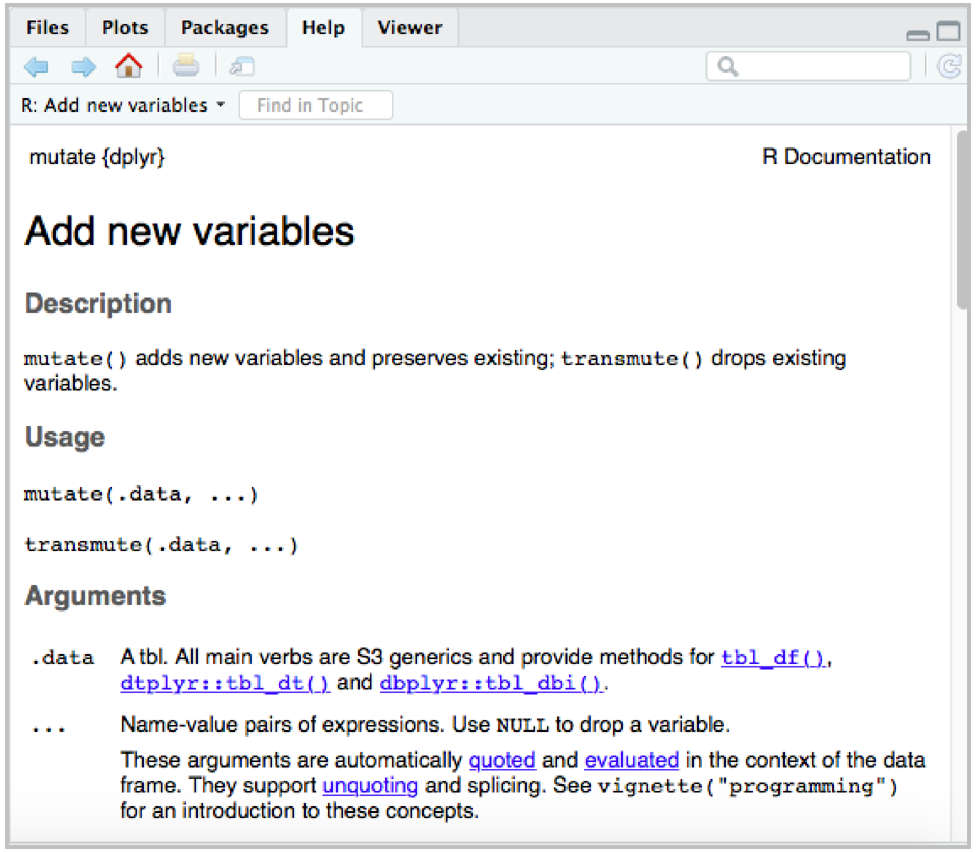
```
# A tibble: 336,776 x 9
    year month   day dep_delay arr_delay distance air_time  gain speed
   <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl> <dbl> <dbl>
 1  2013     1     1         2        11     1400      227     9  370.
 2  2013     1     1         4        20     1416      227    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.
 7  2013     1     1        -5        19     1065      158    24  404.
 8  2013     1     1        -3       -14      229       53   -11  259.
 9  2013     1     1        -3        -8      944      140    -5  405.
10  2013     1     1        -2         8      733      138    10  319.
# … with 336,766 more rows
```

Transmute

```
transmute(df,          # <- dataframe
          new_col1,    # <- rule(s) for new column
          ...)
```

- `transmute` is a function that does the same thing as `mutate` except that it will only keep the new columns
- The 1st argument is the dataframe
- The following arguments are the columns that will be created in your new dataframe

REMEMBER: you are isolating only these new columns


Files Plots Packages Help Viewer

R: Add new variables Find in Topic

mutate {dplyr} R Documentation

Add new variables

Description

`mutate()` adds new variables and preserves existing; `transmute()` drops existing variables.

Usage

```
mutate(.data, ...)

transmute(.data, ...)
```

Arguments

`.data`	A tbl. All main verbs are S3 generics and provide methods for `tbl_df()`, `dtplyr::tbl_dt()` and `dbplyr::tbl_dbi()`.
`...`	Name-value pairs of expressions. Use `NULL` to drop a variable. These arguments are automatically quoted and evaluated in the context of the data frame. They support unquoting and splicing. See `vignette("programming")` for an introduction to these concepts.

Transmute example

- Only returns the new columns

```
# Add two columns `gain` and `speed` to `flights_sml`.
example = transmute(flights_sml,            #<- specify the dataframe
       gain = arr_delay - dep_delay,        #<- create `gain` column by subtracting departure delay
                                            #   from arrival delay
       speed = distance / air_time * 60)    #<- create `speed` from distance and air time columns
example
```

```
# A tibble: 336,776 x 2
    gain speed
   <dbl> <dbl>
 1     9  370.
 2    16  374.
 3    31  408.
 4   -17  517.
 5   -19  394.
 6    16  288.
 7    24  404.
 8   -11  259.
 9    -5  405.
10    10  319.
# … with 336,766 more rows
```

Useful functions for mutate and transmute

When creating new variables with `mutate`, there are many helpful functions that can assist in creating interesting features:

Useful Functions	Explanation
`+, -, *, /, ^`	all mathematic operators can be used on variables
`log, log2, log10`	logarithmic functions for variable transformation can be used
`%/%` and `%%`	modulus and remainder are useful when converting time
`lag(x)` and `lead(x)`	lag and lead allow reference to leading or lagging values - useful for detecting changes in values
`cumsum(x),cummean(x),cummax(x), cumprod(x)`	cumulative, running functions, mins, max, prod, mean, etc.

Useful functions for mutate & transmute cont'd

```
?dplyr::ranking

rank_function(x)  #<- a rank function with
                  #   vector of values
```

- Ranking functions are very helpful; there are several within the `dplyr` package that you can use

Files Plots Packages Help Viewer

R: Windowed rank functions. Find in Topic

Usage

```
row_number(x)

ntile(x, n)

min_rank(x)

dense_rank(x)

percent_rank(x)

cume_dist(x)
```

Arguments

`x`	a vector of values to rank. Missing values are left as is. If you want to treat them as the smallest or largest values, replace with Inf or -Inf before ranking.
`n`	number of groups to split up into.

Details

- `row_number()`: equivalent to `rank(ties.method = "first")`
- `min_rank()`: equivalent to `rank(ties.method = "min")`
- `dense_rank()`: like `min_rank()`, but with no gaps between ranks
- `percent_rank()`: a number between 0 and 1 computed by rescaling `min_rank` to `[0, 1]`
- `cume_dist()`: a cumulative distribution function. Proportion of all values less than or equal to the current rank.
- `ntile()`: a rough rank, which breaks the input vector into `n` buckets.

Exercise 2

Module completion checklist

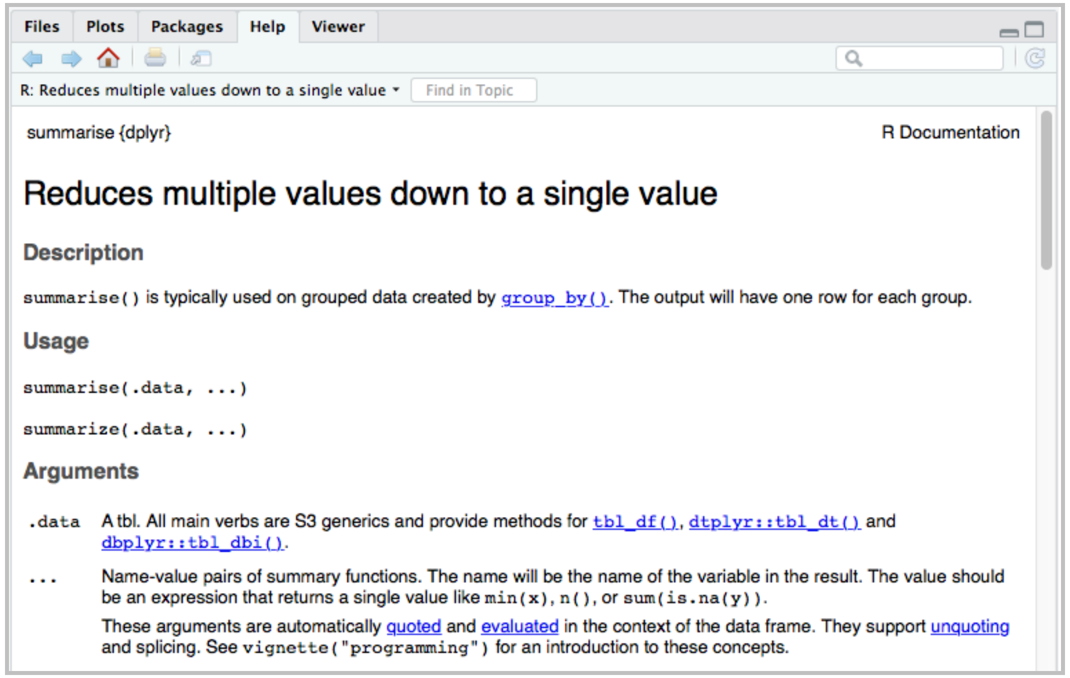
Objective	Complete
Rank data using the arrange function	✔
Select specific variables, sometimes using specific rules, using the select command	✔
Derive new variables from the existing variables using the mutate and transmute commands	✔
Perform multiple functions with the pipe operator (%>%)	
Summarize columns using the summary and group by functions	
Convert wide to long data using tidyr package	
Manipulate columns by using the separate and unite functions	

Summarise and group_by

```
?dplyr::summarise

summarise(df,               #<- dataframe
          summary_func1,    #<- summary rule(s)
                            #    for new col
          ...)
```

- `summarise` collapses a dataframe to a single row
- By itself, `summarise` is not very helpful. We will usually use it with `group_by`

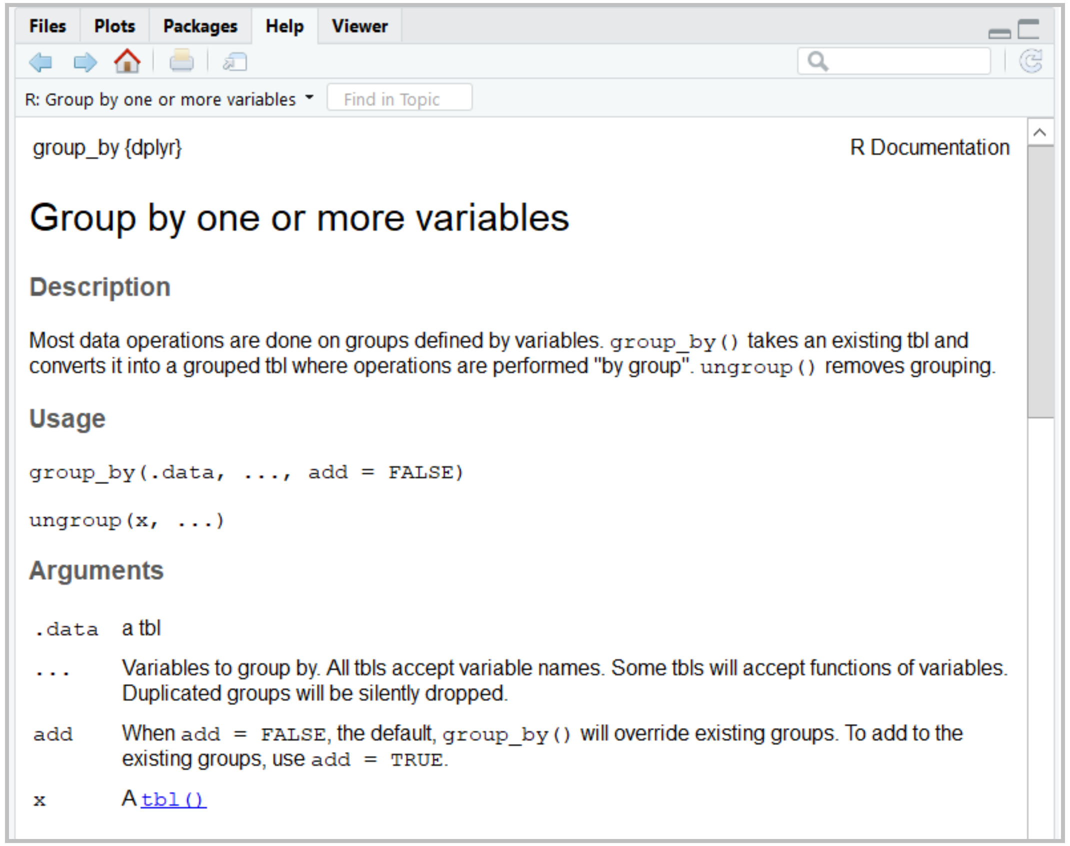


Understanding grouping

```
?dplyr::group_by

group_by(df,          #<- dataframe
         variable1,   #<- 1st var to group by
         variable2,   #<- 2nd var to group by
         ...)
```

- Grouping doesn't change how the data looks (apart from listing how it's grouped).
- It changes how it acts with the other dplyr verbs.
- To removing grouping, use `ungroup`.



Summarise and group_by alone

```
# Produce a summary.
summarise(flights, delay = mean(dep_delay, na.rm = TRUE))
```

```
# A tibble: 1 x 1
  delay
  <dbl>
1  12.6
```

```
# Create `by_day` by grouping `flights` by year, month, and day.
by_day = group_by(flights, year, month, day)
by_day
```

```
# A tibble: 336,776 x 19
# Groups:   year, month, day [365]
    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 336,766 more rows, and 12 more variables: sched_arr_time <int>,
#   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
```

Summarise and group_by together

```r
# Now use grouped `by_day` data and summarise it to
# see the average delay by year, month and day.
summarise(by_day, delay = mean(dep_delay, na.rm = TRUE))
```

```
# A tibble: 365 x 4
# Groups:   year, month [12]
    year month   day delay
   <int> <int> <int> <dbl>
 1  2013     1     1 11.5
 2  2013     1     2 13.9
 3  2013     1     3 11.0
 4  2013     1     4  8.95
 5  2013     1     5  5.73
 6  2013     1     6  7.15
 7  2013     1     7  5.42
 8  2013     1     8  2.55
 9  2013     1     9  2.28
10  2013     1    10  2.84
# … with 355 more rows
```

- `summarise` and `group_by` are two of the most used functions within dplyr!

Dplyr and the pipe: without it

Now we get to the best part, connecting it all. Let's say we want to do these three things:

1. Group flights by destination
2. Summarise to compute distance, average delay, and number of flights
3. Filter to remove noisy points and Honolulu airport, which is almost twice as far away as the next closest airport

We might think we have to write out a `dplyr` function for each, save each as a variable and then perform the next function, which should look something like this:

```
# Step 1: Create a new grouped data
frame `by_dest`.
by_dest = group_by(flights, dest)

      # Step 2: Create a summary of `by_dest` and save as `delay`.
      delay = summarise(by_dest,
                        count = n(),
                        dist = mean(distance, na.rm = TRUE),
                        delay = mean(arr_delay, na.rm = TRUE))

            # Step 3: Filter `delay` by their count and destination.
            delay = filter(delay, count > 20, dest != "HNL")
```

Dplyr and the pipe: a better way

- Sure, that works, but can we do it cleaner? Faster? - **YES!**
- We can use the pipe operator (i.e. `%>%`) and do it all in a single step without creating extra variables

```
delays = flights %>%                                      #<- take flights data
  group_by(dest) %>%                                      #<- group it by destination
  summarise(count = n(),                                  #<- then summarize by creating count variable
            dist = mean(distance, na.rm = TRUE),          #<- and computing mean distance
            delay = mean(arr_delay, na.rm = TRUE)) %>%    #<- and mean arrival delay
  filter(count > 20, dest != "HNL")                       #<- then filter it
delays
```

```
# A tibble: 96 x 4
   dest  count  dist delay
   <chr> <int> <dbl> <dbl>
 1 ABQ     254 1826   4.38
 2 ACK     265  199   4.85
 3 ALB     439  143  14.4
 4 ATL   17215  757. 11.3
 5 AUS    2439 1514.  6.02
 6 AVL     275  584.  8.00
 7 BDL     443  116   7.05
 8 BGR     375  378   8.03
 9 BHM     297  866. 16.9
10 BNA    6333  758. 11.8
# … with 86 more rows
```

Module completion checklist

Objective	Complete
Rank data using the arrange function	✔
Select specific variables, sometimes using specific rules, using the select command	✔
Derive new variables from the existing variables using the mutate and transmute commands	✔
Perform multiple functions with the pipe operator (%>%)	✔
Summarize columns using the summary and group by functions	
Convert wide to long data using tidyr package	
Manipulate columns by using the separate and unite functions	

Summarise and handling NAs

We do NOT address NAs

```
flights %>%
  group_by(year, month, day) %>%
  summarise(mean = mean(dep_delay))
```

```
# A tibble: 365 x 4
# Groups:   year, month [12]
    year month   day  mean
   <int> <int> <int> <dbl>
 1  2013     1     1    NA
 2  2013     1     2    NA
 3  2013     1     3    NA
 4  2013     1     4    NA
 5  2013     1     5    NA
 6  2013     1     6    NA
 7  2013     1     7    NA
 8  2013     1     8    NA
 9  2013     1     9    NA
10  2013     1    10    NA
# … with 355 more rows
```

- If we do not address NAs, the aggregation functions will return NAs for each item if there is just one NA in the input

We address NAs

```
flights %>%
  group_by(year, month, day) %>%
  summarise(mean = mean(dep_delay,
                        na.rm = TRUE))
```

```
# A tibble: 365 x 4
# Groups:   year, month [12]
    year month   day  mean
   <int> <int> <int> <dbl>
 1  2013     1     1 11.5
 2  2013     1     2 13.9
 3  2013     1     3 11.0
 4  2013     1     4  8.95
 5  2013     1     5  5.73
 6  2013     1     6  7.15
 7  2013     1     7  5.42
 8  2013     1     8  2.55
 9  2013     1     9  2.28
10  2013     1    10  2.84
# … with 355 more rows
```

- Moral of the story: remember to address NAs when using summarise!

A few more useful summary functions

Summary Functions	Explanation
`n()`	Will count the number of entries that come from a summarise
`min(x),quantile(x, 0.25),max(x)`	Measures of rank and distribution can be used
`first(x),nth(x, 2),last(x)`	Measures of position and order
`n_distinct`	Will count the number of distinct values

Summarise n to count

`n` will count the number of entries that come from a `summarise` function

```
flights %>%
  group_by(year, month, day) %>%
  summarise(mean = mean(dep_delay, na.rm = TRUE),
            n = n()) #<- add a column with summary counts
```

```
# A tibble: 365 x 5
# Groups:   year, month [12]
    year month   day  mean     n
   <int> <int> <int> <dbl> <int>
 1  2013     1     1 11.5    842
 2  2013     1     2 13.9    943
 3  2013     1     3 11.0    914
 4  2013     1     4  8.95   915
 5  2013     1     5  5.73   720
 6  2013     1     6  7.15   832
 7  2013     1     7  5.42   933
 8  2013     1     8  2.55   899
 9  2013     1     9  2.28   902
10  2013     1    10  2.84   932
# … with 355 more rows
```

Actually, we do not need summarise to count

`count` is a function to just count, even if you have not used a summary function

```
flights %>%
  count(day) #<- count number of instances of entry in `day` column
```

```
# A tibble: 31 x 2
     day     n
   <int> <int>
 1     1 11036
 2     2 10808
 3     3 11211
 4     4 11059
 5     5 10858
 6     6 11059
 7     7 10985
 8     8 11271
 9     9 10857
10    10 11227
# … with 21 more rows
```

Summarise rank

Measures of rank: `min(x)`, `quantile(x, 0.25)`, `max(x)`

```
flights %>%
  group_by(year, month) %>%
  summarise(first = min(dep_time, na.rm = TRUE),
            last = max(dep_time, na.rm = TRUE))
```

```
# A tibble: 12 x 4
# Groups:   year [1]
    year month first  last
   <int> <int> <dbl> <dbl>
 1  2013     1     1  2359
 2  2013     2     1  2400
 3  2013     3     1  2400
 4  2013     4     1  2400
 5  2013     5     1  2400
 6  2013     6     1  2400
 7  2013     7     1  2400
 8  2013     8     1  2400
 9  2013     9     2  2400
10  2013    10     6  2400
11  2013    11     1  2400
12  2013    12     1  2400
```

Summarise position

```
# 1. Build a subset of all flights that were not cancelled.
not_cancelled = flights %>%
  filter(!is.na(dep_time))  #<- filter flights where `dep_time` was not `NA`

# 2. Group and summarize all flights that were not cancelled to get desired results.
not_cancelled  %>%
  group_by(year, month, day) %>%   #<- group the not cancelled flights
  summarise(first = min(dep_time), #<- then summarise them by calculating the first
            last = max(dep_time))  #<- and last flights in the `dep_time` in each group
```

```
# A tibble: 365 x 5
# Groups:   year, month [12]
    year month   day first  last
   <int> <int> <int> <dbl> <dbl>
 1  2013     1     1   517  2356
 2  2013     1     2    42  2354
 3  2013     1     3    32  2349
 4  2013     1     4    25  2358
 5  2013     1     5    14  2357
 6  2013     1     6    16  2355
 7  2013     1     7    49  2359
 8  2013     1     8   454  2351
 9  2013     1     9     2  2252
10  2013     1    10     3  2320
# … with 355 more rows
```

Summarise distinct values

`n_distinct(x)` will count the number of distinct values

```
# Number of flights that take off, by day.
not_cancelled  %>%
  group_by(year, month, day) %>%
  summarise(flights_that_take_off = n_distinct(dep_time)) #<- calculate distinct departure times
```

```
# A tibble: 365 x 4
# Groups:   year, month [12]
    year month   day flights_that_take_off
   <int> <int> <int>                 <int>
 1  2013     1     1                   552
 2  2013     1     2                   583
 3  2013     1     3                   589
 4  2013     1     4                   589
 5  2013     1     5                   495
 6  2013     1     6                   564
 7  2013     1     7                   572
 8  2013     1     8                   573
 9  2013     1     9                   580
10  2013     1    10                   572
# … with 355 more rows
```

Remember to ungroup before you regroup

```
# Take the same `not_cancelled` data, but now group by month instead of by day.
not_cancelled %>%                                            #<- set dataframe
  ungroup() %>%                                              #<- first ungroup it
  group_by(year, month) %>%                                  #<- then group by year and month
  summarise(flights_by_year = n_distinct(dep_time))#<- then do the rest ...
```

```
# A tibble: 12 x 3
# Groups:   year [1]
    year month flights_by_year
   <int> <int>           <int>
 1  2013     1            1165
 2  2013     2            1171
 3  2013     3            1199
 4  2013     4            1216
 5  2013     5            1186
 6  2013     6            1220
 7  2013     7            1242
 8  2013     8            1204
 9  2013     9            1156
10  2013    10            1139
11  2013    11            1135
12  2013    12            1191
```

Knowledge check 2

Exercise 3

Module completion checklist

Objective	Complete
Rank data using the arrange function	✔
Select specific variables, sometimes using specific rules, using the select command	✔
Derive new variables from the existing variables using the mutate and transmute commands	✔
Perform multiple functions with the pipe operator (%>%)	✔
Summarize columns using the summary and group by functions	✔
Convert wide to long data using tidyr package	
Manipulate columns by using the separate and unite functions	

Defining data wrangling

- Data transformation prepares the dataset for wrangling
- We want all the variables and values, all the new columns to be created, and all the NAs taken care of before making sure it is in `tidy` form
- `tidyr`, the package within `tidyverse`, allows us to get our data into tidy format!
- We will use the `tidyr_tables.Rdata` file that we loaded at the beginning of the lesson

- For further reading and understanding of tidy data and where it originated, check out this link: ***http://www.jstatsoft.org/v59/i10/paper***

Would analysis of these datasets be easy?

```
key_value_country
```

```
# A tibble: 12 x 4
   country      year key              value
   <fct>       <int> <fct>            <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
```

```
year_country
```

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

```
rate_country
```

```
# A tibble: 6 x 3
  country      year rate
  <fct>       <int> <chr>
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
```

What makes data 'tidy'?

- Three interrelated rules make a dataset tidy:
  - Each variable must have its own column
  - Each observation must have its own row
  - Each value must have its own cell

- `tidy_country` is the only table that follows all 3 rules

```
tidy_country
```

```
# A tibble: 6 x 4
  country      year  cases population
  <fct>       <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
```

What are the advantages of tidy data?

- Consistent way of storing data: it's easier to learn the tools that work with it because of the underlying uniformity
- Making use of R's internal vectorization: most built-in R functions work with vectors of values
- Making use of `spread` and `gather`: the functions of `tidyr` that will help you transform `messy` data to `tidy` data

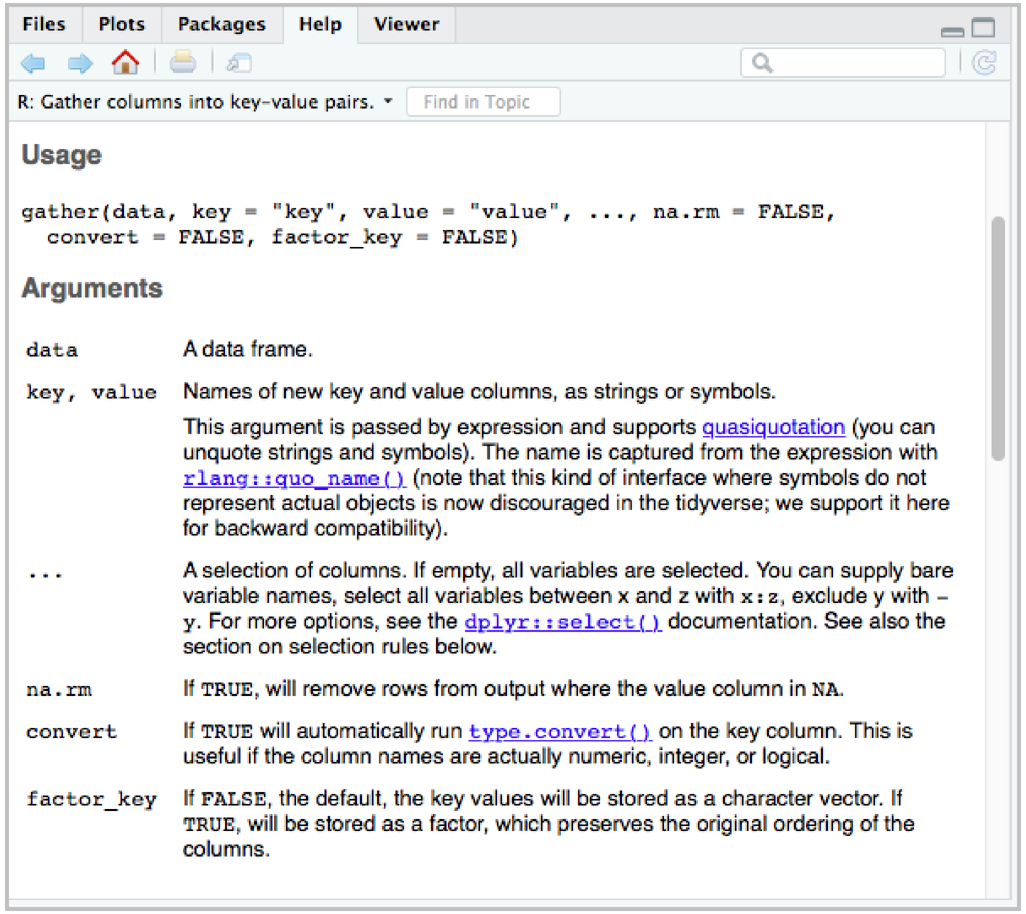
Gathering

- `gather` pulls multiple columns into one new variable

```
?dplyr::gather

gather(df,      #<- dataframe
       key,     #<- name of new key column
       value )  #<- name of new value column
```

- We need three parameters to describe the operation of `gather`:
  - The columns that represent the values
  - The name of the variable (the `key`) that represents those values
  - The name of the variable (the `value`) that represents the values that are currently within the value columns



Usage

```
gather(data, key = "key", value = "value", ..., na.rm = FALSE,
  convert = FALSE, factor_key = FALSE)
```

Arguments

`data`	A data frame.
`key, value`	Names of new key and value columns, as strings or symbols. This argument is passed by expression and supports quasiquotation (you can unquote strings and symbols). The name is captured from the expression with `rlang::quo_name()` (note that this kind of interface where symbols do not represent actual objects is now discouraged in the tidyverse; we support it here for backward compatibility).
`...`	A selection of columns. If empty, all variables are selected. You can supply bare variable names, select all variables between x and z with `x:z`, exclude y with `-y`. For more options, see the `dplyr::select()` documentation. See also the section on selection rules below.
`na.rm`	If `TRUE`, will remove rows from output where the value column in `NA`.
`convert`	If `TRUE` will automatically run `type.convert()` on the key column. This is useful if the column names are actually numeric, integer, or logical.
`factor_key`	If `FALSE`, the default, the key values will be stored as a character vector. If `TRUE`, will be stored as a factor, which preserves the original ordering of the columns.

Gathering problem - colnames as values

```
# Let's look at `year_country`.
year_country
```

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

- Notice that the second and third column are both values, these could be in one variable `year`
- Let's use `gather` to bring the two columns, `1999` and `2000` into one column `year`
- Let's make the second column `cases`, which will contain the count that currently appears in each year's column

Gather function example

```
# Gather the `year_country` dataframe to make it tidy.
year_country %>%           #<- set the dataframe and use pipe to use it as input into `gather`
  gather(`1999`,           #<- set 1st column to gather
         `2000`,           #<- set 2nd column to gather
         key = "year",     #<- set `year` column as a key
         value = "cases")  #<- set `cases` column as the values from the columns we gather
```

```
# A tibble: 6 x 3
  country     year   cases
  <fct>       <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
```

- Remember, the combination of data, function parameters, and the pipe (`%>%`) is common not only to `dplyr`, but also to all of the packages within `tidyverse`!

Gather function: specifying a range

```
# Gather the `year_country` dataframe to make it tidy.
year_country %>%          #<- set the dataframe and use pipe to use it as input into `gather`
  gather(2:3,             #<- provide a range of columns to gather
         key = "year",    #<- set `year` column as a key
         value = "cases") #<- set `cases` column as the values from the columns we gather
```

```
# A tibble: 6 x 3
  country     year   cases
  <fct>       <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
```

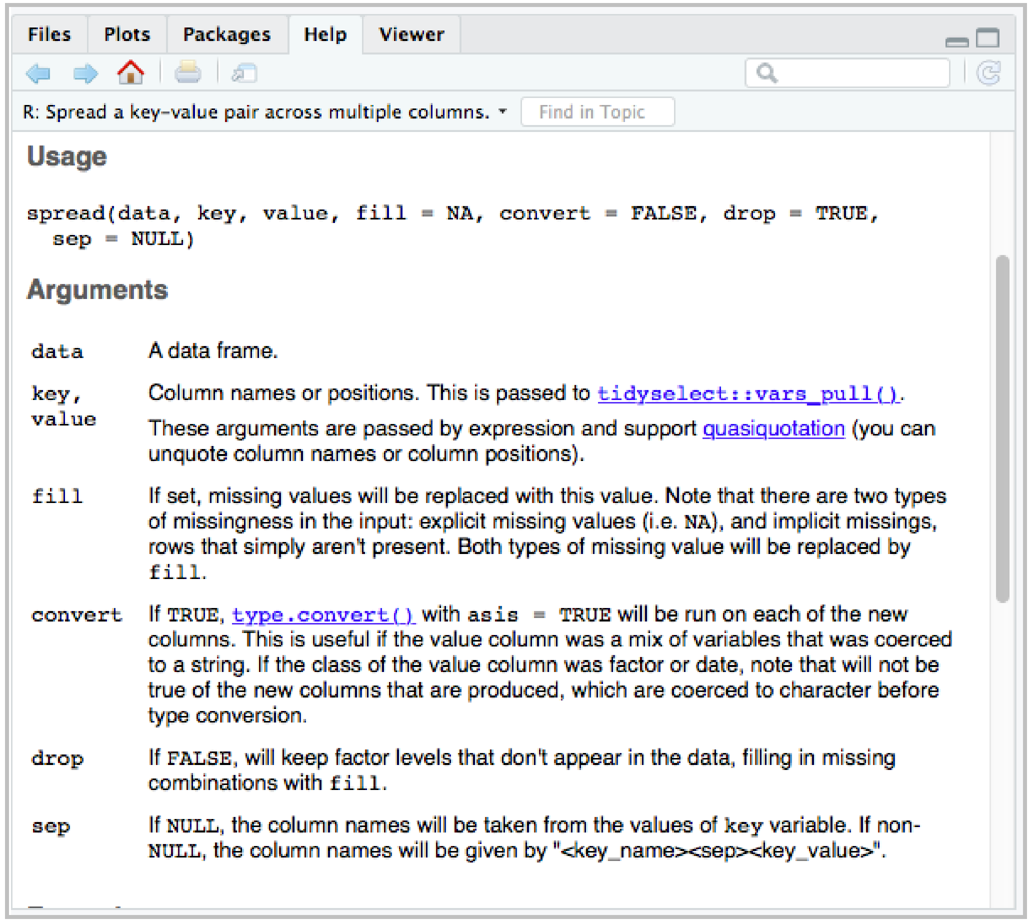
- Note that the code substituted `2:3` rather than the named columns

Spreading

```
?dplyr::spread

spread(df,    #<- dataframe
          key,    #<- name of current key column
          value) #<- name of current value column
```

- `spread` spreads one column into multiple variables
- Spreading is the opposite of gathering
- You use it when an observation is scattered across multiple rows
- There are two parameters we need to pay attention to when using `spread`:
  - The column that contains the variable names, the `key` column
  - The column that contains the values for the multiple variables, the `value` column



Spreading

```
# Let's look at `key_value_country`.
key_value_country
```

```
# A tibble: 12 x 4
   country      year key             value
   <fct>       <int> <fct>           <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
```

- How would we use spread?
- Use `key_value_country` as initial dataframe
- Use `spread` with **2 main parameters:**
  - The `key`, which contains the variables
  - The `value`, which contains the values for each of the rows of the variables in the `key` column

Spread: two ways

```
# Spread the data
# Pass data to spread with pipe.
key_value_country %>%
  spread(key = key,
         value = value)
```

```
# A tibble: 6 x 4
  country      year  cases population
  <fct>       <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 without the pipe.
# Dataframe passed in.
spread(key_value_country,
       key = key,
       value = value)
```

```
# A tibble: 6 x 4
  country      year  cases population
  <fct>       <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
```

Module completion checklist

Objective	Complete
Rank data using the arrange function	✔
Select specific variables, sometimes using specific rules, using the select command	✔
Derive new variables from the existing variables using the mutate and transmute commands	✔
Perform multiple functions with the pipe operator (%>%)	✔
Summarize columns using the summary and group by functions	✔
Convert wide to long data using tidyr package	✔
Manipulate columns by using the separate and unite functions	

Defining Separating and uniting

- How would we adjust a single variable?
- What would we use for a dataframe like `rate_country`?

```
rate_country
```

```
# A tibble: 6 x 3
  country      year rate
  <fct>       <int> <chr>
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
```

- What do we do with the `rate` column?
- We can use the function `separate`
- The opposite of `separate` is `unite`
- We will learn how to use this as well

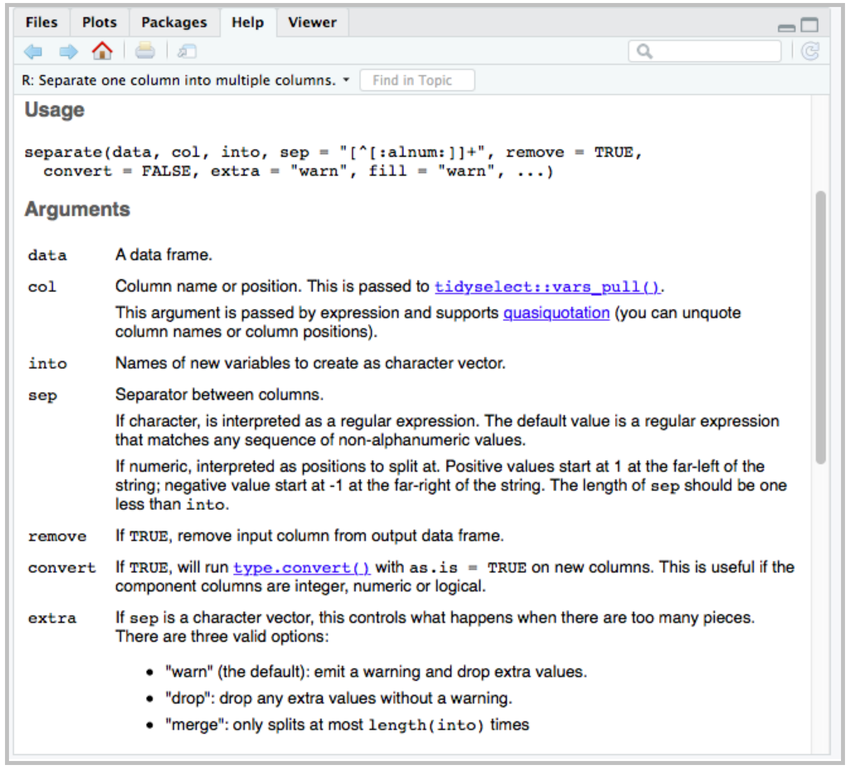
Separate

```
?dplyr::separate

separate(df,    #<- dataframe
         col,   #<- name of column to separate
         into)  #<- name of new variables to
                #   create as a character vector
```

- `separate` divides a single character column into multiple columns and takes two arguments:
- The first argument is the dataframe
- Next we pipe it to `separate`

1. The first parameter is the column to be separated
2. The second is what we want to separate the variable into, using `into = c("var_1", "var_2")`


Files Plots Packages Help Viewer

R: Separate one column into multiple columns. Find in Topic

Usage

```
separate(data, col, into, sep = "[^[:alnum:]]+", remove = TRUE,
  convert = FALSE, extra = "warn", fill = "warn", ...)
```

Arguments

`data`	A data frame.
`col`	Column name or position. This is passed to `tidyselect::vars_pull()`. This argument is passed by expression and supports quasiquotation (you can unquote column names or column positions).
`into`	Names of new variables to create as character vector.
`sep`	Separator between columns. If character, is interpreted as a regular expression. The default value is a regular expression that matches any sequence of non-alphanumeric values. If numeric, interpreted as positions to split at. Positive values start at 1 at the far-left of the string; negative value start at -1 at the far-right of the string. The length of `sep` should be one less than `into`.
`remove`	If `TRUE`, remove input column from output data frame.
`convert`	If `TRUE`, will run `type.convert()` with `as.is = TRUE` on new columns. This is useful if the component columns are integer, numeric or logical.
`extra`	If `sep` is a character vector, this controls what happens when there are too many pieces. There are three valid options: • "warn" (the default): emit a warning and drop extra values. • "drop": drop any extra values without a warning. • "merge": only splits at most `length(into)` times

Separate

```
# Using `rate_country`, separate the `rate` column into two.
rate_country %>%                     #<- set dataframe and pass it to next function with pipe
  separate(rate,                     #<- separate `rate`
           into = c("cases",         #<- into column `cases`, and
                    "population")) #<-      column `population`
```

```
# A tibble: 6 x 4
  country      year cases  population
  <fct>       <int> <chr>  <chr>
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
```

Separate

- By default, `separate` will separate on any non alpha-numeric character
- However, you can also specify the character to separate on

```
# Using `rate_country`, separate the `rate` column into two.
rate_country %>%
  separate(rate,
           into = c("cases",
                    "population"),
           sep = "/")               #<- set the separating character to `/`
```

```
# A tibble: 6 x 4
  country      year cases  population
  <fct>       <int> <chr>  <chr>
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
```

Separate: sep set to index

- You can use the `sep` parameter to separate the year column on the character index into `century` and `year`

```
# Using `rate_country`, separate the `year` column into two.
rate_country %>%
  separate(year,                  #<- separate `year`
           into= c("century",     #<- into two columns: `century`, and
                   "year"),       #<- `year`
           sep = 2)               #<- set the separator at index = 2
```

```
# A tibble: 6 x 4
  country     century year  rate
  <fct>       <chr>   <chr> <chr>
1 Afghanistan 19      99    745/19987071
2 Afghanistan 20      00    2666/20595360
3 Brazil      19      99    37737/172006362
4 Brazil      20      00    80488/174504898
5 China       19      99    212258/1272915272
6 China       20      00    213766/1280428583
```

Separate: data type conversion

When we use `separate`, the data type of the original column will be preserved

```
# The new columns are now also characters.
rate_country %>%
  separate(rate, into = c("cases", "population"))
```

```
# A tibble: 6 x 4
  country      year cases  population
  <fct>       <int> <chr>  <chr>
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
```

However, we can tell `separate` to convert to what it thinks the new columns should be

```
rate_country %>%
  separate(rate, into = c("cases", "population"), convert = TRUE)
```

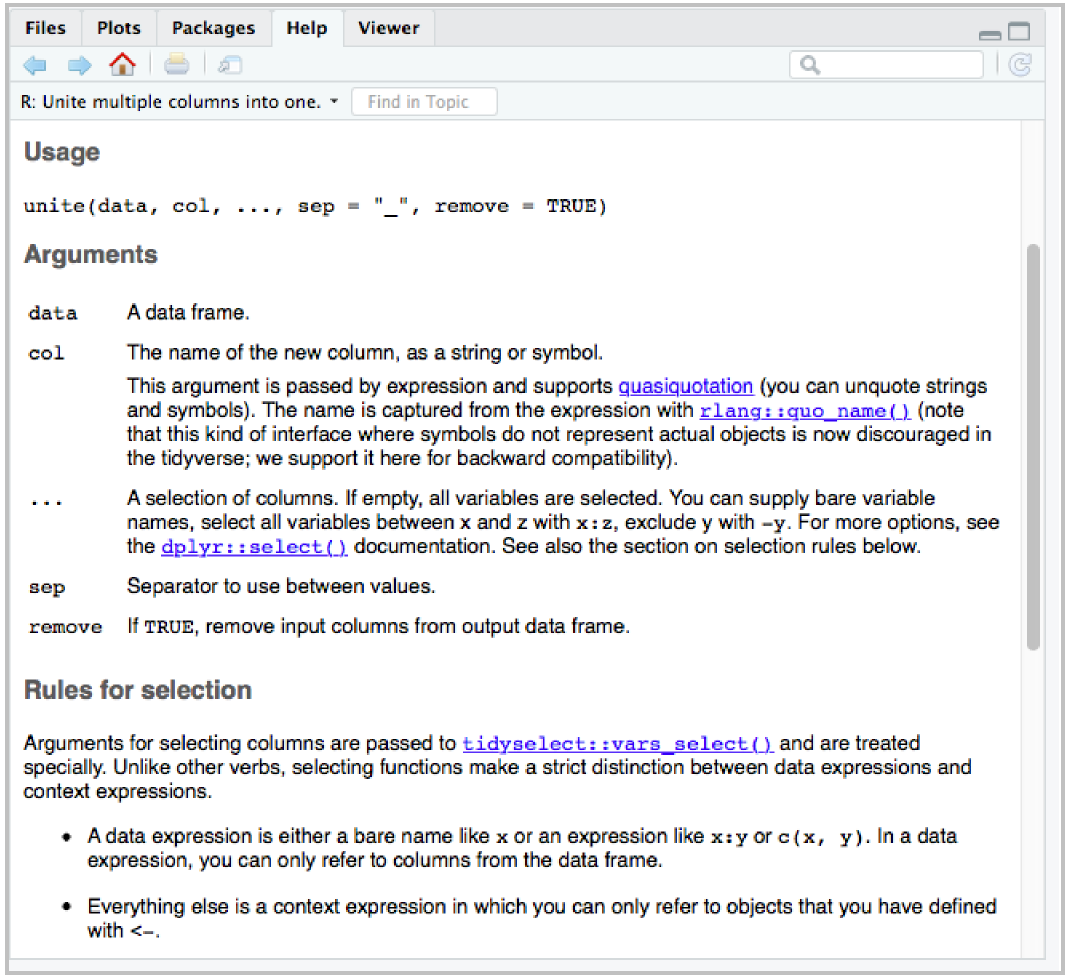
```
# A tibble: 6 x 4
  country      year  cases population
  <fct>       <int>  <int>      <int>
1 Afghanistan  1999    745   19987071
2 Afghanistan  2000   2666   20595360
3 Brazil       1999  37737  172006362
4 Brazil       2000  80488  174504898
5 China        1999 212258 1272915272
6 China     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

Unite

```
?dplyr::unite

unite(df,  #<- dataframe
      col, #<- name of column to unite
      sep) #<- separator to use
```

- `unite` combines multiple character columns into a single one
- `unite` is the inverse of separate



Files Plots Packages Help Viewer

R: Unite multiple columns into one. Find in Topic

Usage

```
unite(data, col, ..., sep = "_", remove = TRUE)
```

Arguments

`data`	A data frame.
`col`	The name of the new column, as a string or symbol. This argument is passed by expression and supports quasiquotation (you can unquote strings and symbols). The name is captured from the expression with `rlang::quo_name()` (note that this kind of interface where symbols do not represent actual objects is now discouraged in the tidyverse; we support it here for backward compatibility).
`...`	A selection of columns. If empty, all variables are selected. You can supply bare variable names, select all variables between x and z with `x:z`, exclude y with `-y`. For more options, see the `dplyr::select()` documentation. See also the section on selection rules below.
`sep`	Separator to use between values.
`remove`	If `TRUE`, remove input columns from output data frame.

Rules for selection

Arguments for selecting columns are passed to `tidyselect::vars_select()` and are treated specially. Unlike other verbs, selecting functions make a strict distinction between data expressions and context expressions.

- A data expression is either a bare name like `x` or an expression like `x:y` or `c(x, y)`. In a data expression, you can only refer to columns from the data frame.
- Everything else is a context expression in which you can only refer to objects that you have defined with `<-`.

Unite example

We will use the separated-on-year example of `rate_country` to show `unite`

```
# Let's separate the `rate_country`'s `year` column into `century` and `year` first.
ex_table = rate_country %>%
  separate(year, into = c("century", "year"), sep = 2, convert = TRUE)

# Now we use `unite` to combine the two new columns back into one.
# By default, unite will combine columns using `_` so we can use `sep` to specify that we
# do not want anything between the two columns when combined into one cell.
ex_table %>%        #<- specify the dataframe to pipe into `unite`
  unite(time,       #<- set the column `time` for combined values
        century,    #<- 1st column to unite
        year,       #<- 2nd column to unite
        sep = "")   #<- set the separator to an empty string
```

```
# A tibble: 6 x 3
  country     time  rate
  <fct>       <chr> <chr>
1 Afghanistan 1999  745/19987071
2 Afghanistan 200   2666/20595360
3 Brazil      1999  37737/172006362
4 Brazil      200   80488/174504898
5 China       1999  212258/1272915272
6 China       200   213766/1280428583
```

Knowledge check 3

Exercise 4

Module completion checklist

Objective	Complete
Rank data using the arrange function	✔
Select specific variables, sometimes using specific rules, using the select command	✔
Derive new variables from the existing variables using the mutate and transmute commands	✔
Perform multiple functions with the pipe operator (%>%)	✔
Summarize columns using the summary and group by functions	✔
Convert wide to long data using tidyr package	✔
Manipulate columns by using the separate and unite functions	✔

Workshop!

- **Today will be your first *after class* workshop**
- Workshops are to be completed outside of class and emailed to the instructor by the beginning of class tomorrow
- Make sure to comment your code so that it is easy for others to understand what you are doing
- This is an exploratory exercise to get you comfortable with the content we discussed today
- Workshop objectives:
  - Practice different `tidyverse` functions on the dataset you have chosen in the last class
  - Clean the dataset
  - Create new variables using mutate functions as you seem fit

Congratulations on completing the module!