

DATA SOCIETY®

Week 2 Day 1 - Fundamentals of R Part 2

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

Module completion checklist

Topic	Complete
Demonstrate installing a package and loading a library	
Define the six functions that provide verbs for the language of data manipulation, from the package dplyr	
Apply the filter function to subset data	
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	
Summarize columns using the summary and group by functions	
Convert wide to long data using tidyr package	
Discuss 3 reasons of why it is important to transform and wrangle data before moving forward with analyses	
Manipulate columns by using the separate and unite functions	

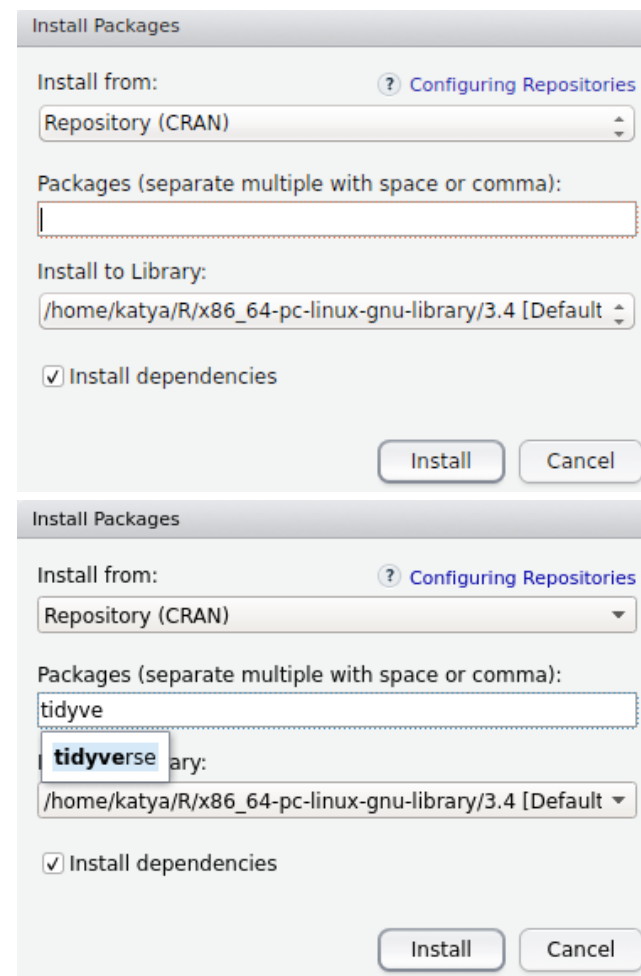
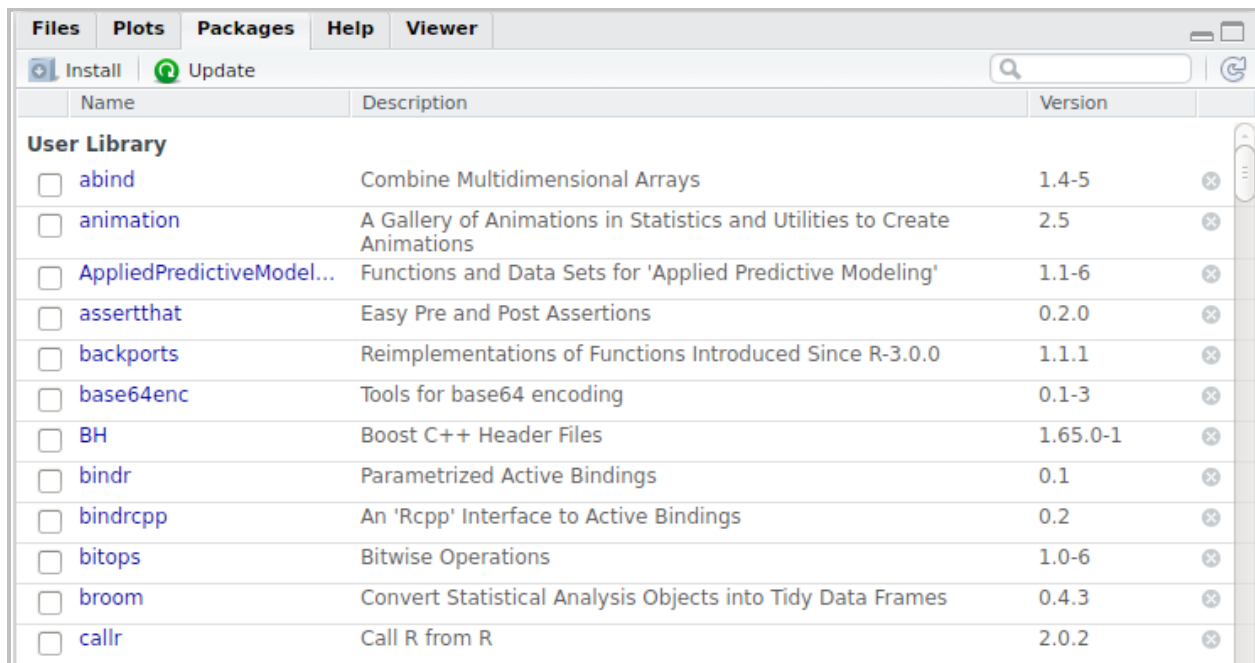
Installing packages: package explorer

RStudio has a built-in package manager in the bottom right pane to help us install packages

Click on **Packages** tab in the **bottom-right** pane

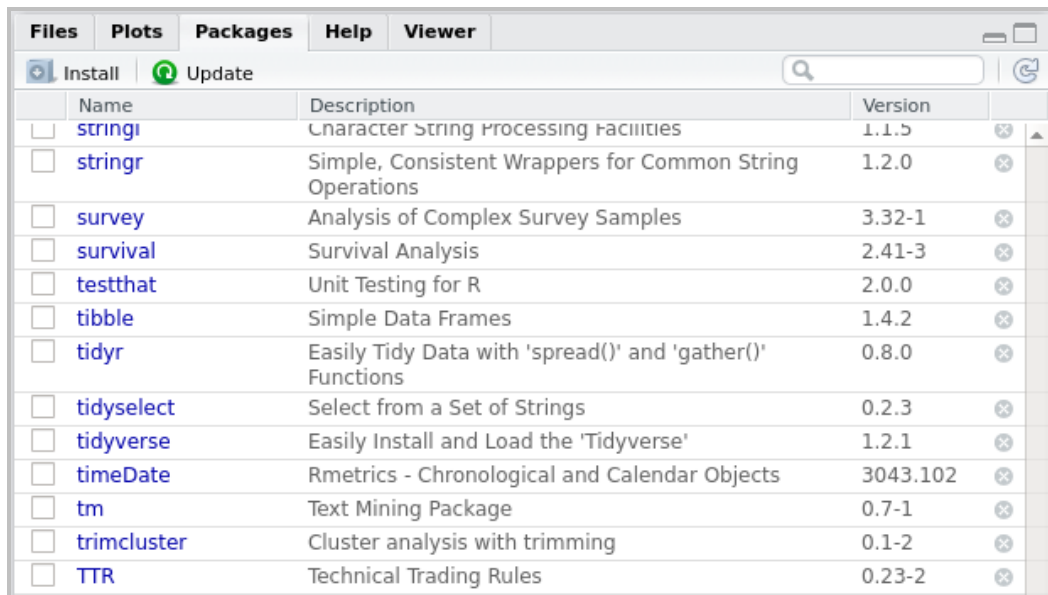
Click **Install** button next to Update

Type package name in the box and install



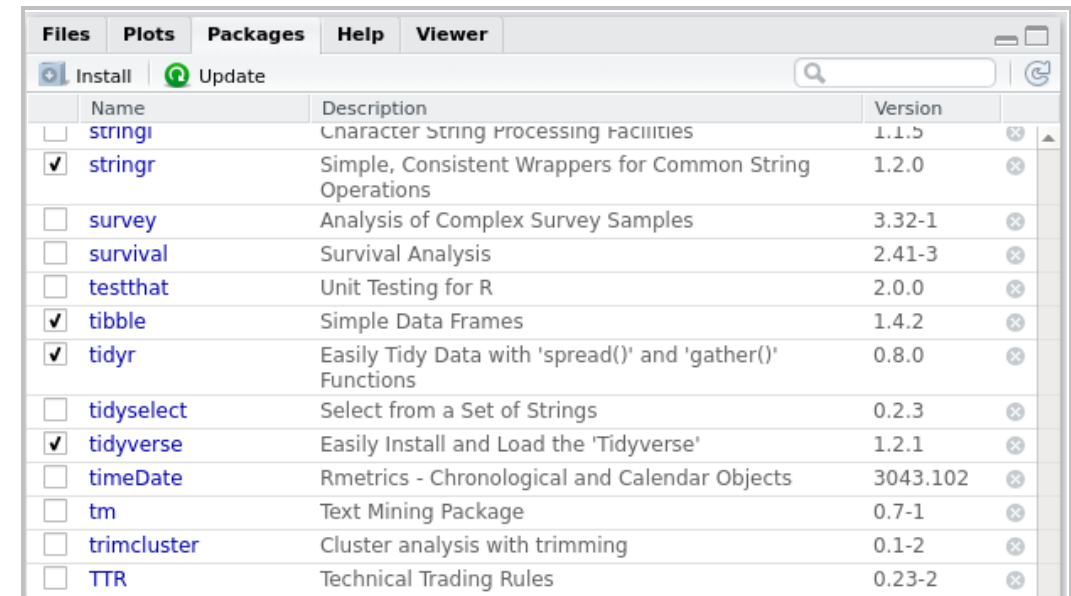
Installing packages: package explorer

The installed package should appear in the list of packages in the package explorer



Files	Plots	Packages	Help	Viewer
Install Update				
Name	Description	Version		
<input type="checkbox"/> stringi	Character String Processing Facilities	1.1.5	⊗	
<input type="checkbox"/> stringr	Simple, Consistent Wrappers for Common String Operations	1.2.0	⊗	
<input type="checkbox"/> survey	Analysis of Complex Survey Samples	3.32-1	⊗	
<input type="checkbox"/> survival	Survival Analysis	2.41-3	⊗	
<input type="checkbox"/> testthat	Unit Testing for R	2.0.0	⊗	
<input type="checkbox"/> tibble	Simple Data Frames	1.4.2	⊗	
<input type="checkbox"/> tidyr	Easily Tidy Data with 'spread()' and 'gather()' Functions	0.8.0	⊗	
<input type="checkbox"/> tidyselect	Select from a Set of Strings	0.2.3	⊗	
<input type="checkbox"/> tidyverse	Easily Install and Load the 'Tidyverse'	1.2.1	⊗	
<input type="checkbox"/> timeDate	Rmetrics - Chronological and Calendar Objects	3043.102	⊗	
<input type="checkbox"/> tm	Text Mining Package	0.7-1	⊗	
<input type="checkbox"/> trimcluster	Cluster analysis with trimming	0.1-2	⊗	
<input type="checkbox"/> TTR	Technical Trading Rules	0.23-2	⊗	

To load the package into R's environment, check the box next to the name of your desired package



Files	Plots	Packages	Help	Viewer
Install Update				
Name	Description	Version		
<input type="checkbox"/> stringi	Character String Processing Facilities	1.1.5	⊗	
<input checked="" type="checkbox"/> stringr	Simple, Consistent Wrappers for Common String Operations	1.2.0	⊗	
<input type="checkbox"/> survey	Analysis of Complex Survey Samples	3.32-1	⊗	
<input type="checkbox"/> survival	Survival Analysis	2.41-3	⊗	
<input type="checkbox"/> testthat	Unit Testing for R	2.0.0	⊗	
<input checked="" type="checkbox"/> tibble	Simple Data Frames	1.4.2	⊗	
<input checked="" type="checkbox"/> tidyr	Easily Tidy Data with 'spread()' and 'gather()' Functions	0.8.0	⊗	
<input type="checkbox"/> tidyselect	Select from a Set of Strings	0.2.3	⊗	
<input checked="" type="checkbox"/> tidyverse	Easily Install and Load the 'Tidyverse'	1.2.1	⊗	
<input type="checkbox"/> timeDate	Rmetrics - Chronological and Calendar Objects	3043.102	⊗	
<input type="checkbox"/> tm	Text Mining Package	0.7-1	⊗	
<input type="checkbox"/> trimcluster	Cluster analysis with trimming	0.1-2	⊗	
<input type="checkbox"/> TTR	Technical Trading Rules	0.23-2	⊗	

Installing packages

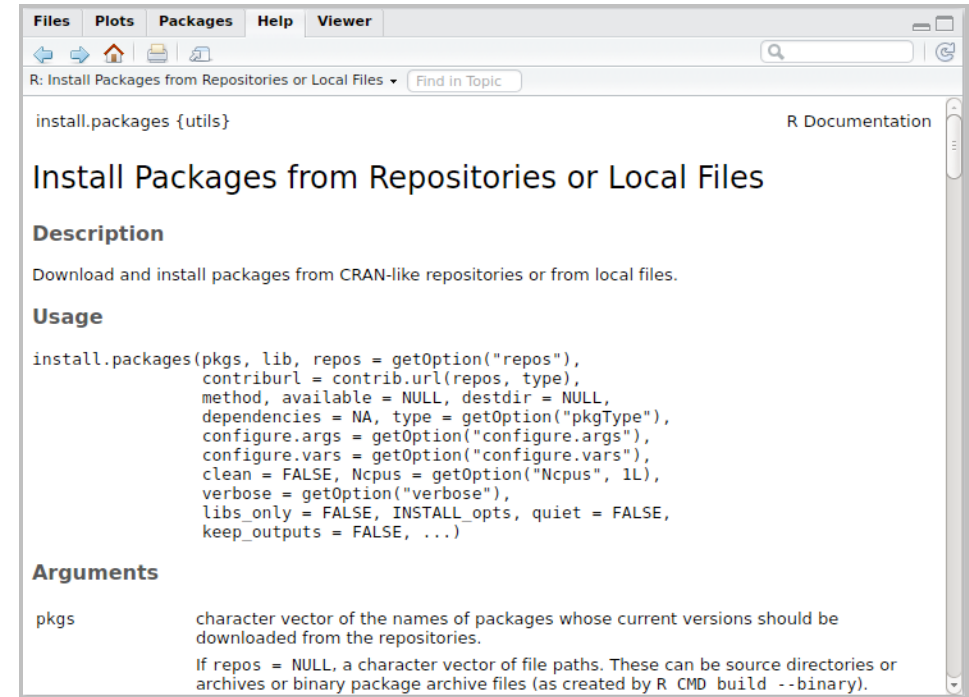
If the function we would like to use comes from a package, we need to **install** the package first

In addition to installing packages with package explorer as we introduced earlier, a more frequent way is to use function

install.packages()

For this function, we need to provide a single required argument: a character string corresponding to the package name

```
# Install package
?install.packages
```

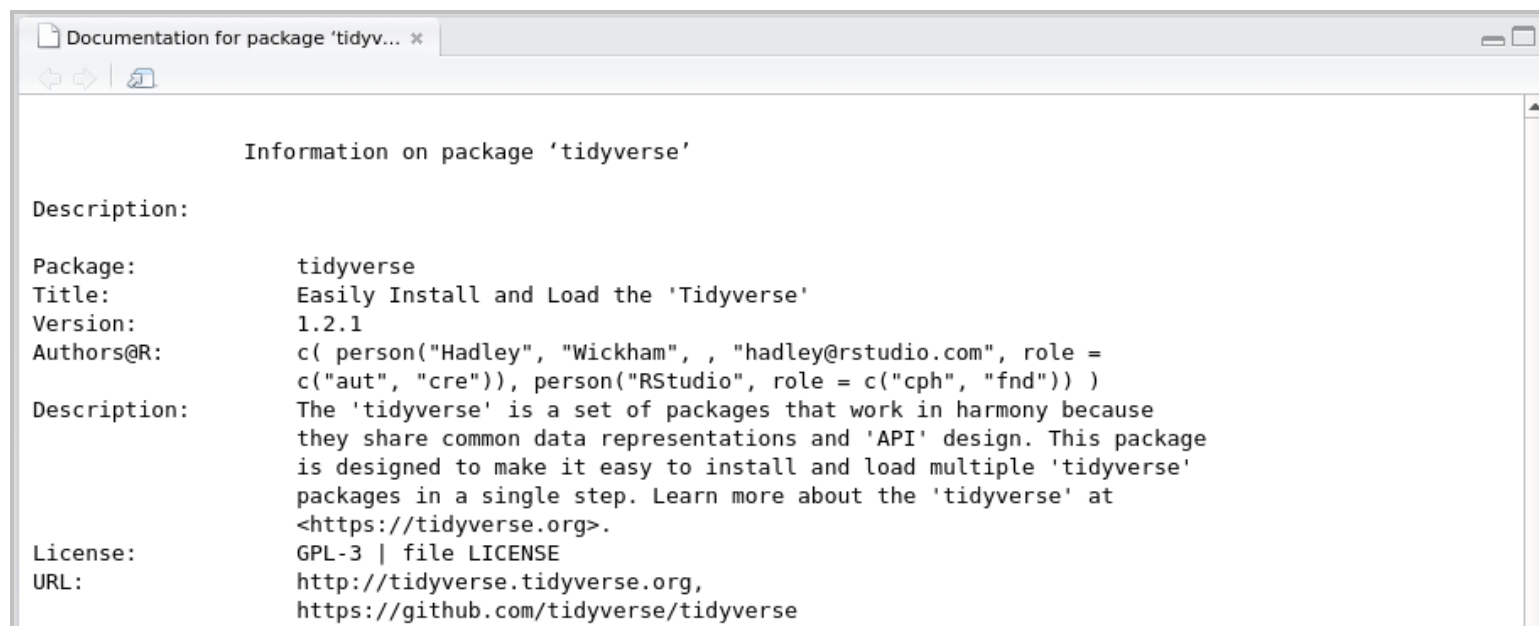


Installing packages

Here is an example of how we install and load packages with function **`install.packages()`**

You can always check the detailed documentation of one package with `help = "package name"`

```
install.packages("tidyverse")    #<- Install package
library(tidyverse)               #<- Load the package into the environment.
library(help = "tidyverse")      #<- View package documentation.
```



Directory settings

In order to maximize the efficiency of your workflow, you may want to encode your directory structure into variables

Let the `main_dir` be the variable corresponding to your `hhs-r-2020` folder

```
# Set `main_dir` to the location of your `hhs-r-2020` folder (for Mac/Linux).
main_dir = "~/Desktop/hhs-r-2020"
# Set `main_dir` to the location of your `hhs-r-2020` folder (for Windows).
main_dir = "C:/Users/[username]/Desktop/hhs-r-2020"

# 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 directory to data_dir.
setwd(data_dir)
```

Installing packages and loading data

To review the functions within various R packages, we will need to import our dataset. R comes with several **built-in** data packages. The following is a list of some of the most common datasets:

- **Titanic**: Survival of passengers on the Titanic
- **iris**: Edgar Anderson's Iris Data
- **mtcars**: Motor Trend Car Road Tests

Today we'll be using one built-in dataset from R called **nycflights13** which describes **airline on-time information for all flights departing NYC in 2013**

Installing packages and loading data

Let's now install and load the `nycflights13` package

```
#install.packages("nycflights13")  
library(nycflights13)
```

The `nycflights13` package contains the following five datasets:

- `flights`: all flights that departed from NYC in 2013
- `weather`: hourly meteorological data for each airport
- `planes`: construction information about each plane
- `airports`: airport names and locations
- `airlines`: translation between two letter carrier codes and names

Installing packages and loading data

.RData is a specific format designed for storing a complete R workspace, or selected objects from a workspace, in a form that can be easily loaded back into R

RData files are organized as a sequence of objects while each object has a type

We will get back to it with a more detailed introduction later in this module

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

Introduction to data transformation with tidyverse

When you are given messy data, your goal is to transform it into a usable format

To do this, you may need the help from multiple **packages** that can be found within the universe of *tidyverse*

Some core packages in *tidyverse* are: *ggplot2*, *dplyr*, *tidyr*

In this module, we will go over how to:

- manipulate data with : *dplyr*
- transform data with: *tidyr*



A little more about tidyverse

Packages in the tidyverse change fairly frequently

You can see if updates are available, and optionally install them, by running the following code

```
tidyverse_update()
```

Like we noted previously, there are many libraries within the `tidyverse` package

The packages we will focus on help you wrangle and manipulate data quickly and efficiently

Data transformation

dplyr is an essential library within the tidyverse universe

It will be the tool we use for transforming our data by filtering, aggregating, and summarizing

Before starting this lesson, understand that dplyr does **overwrite** some base R packages such as `filter` and `lag`

Even functions with exactly the same name can be of different usage and syntax when belonging to different packages

If you have loaded dplyr and want to use the base version of the package, you will have to type in the full name: `stats::filter` and `stats::lag`

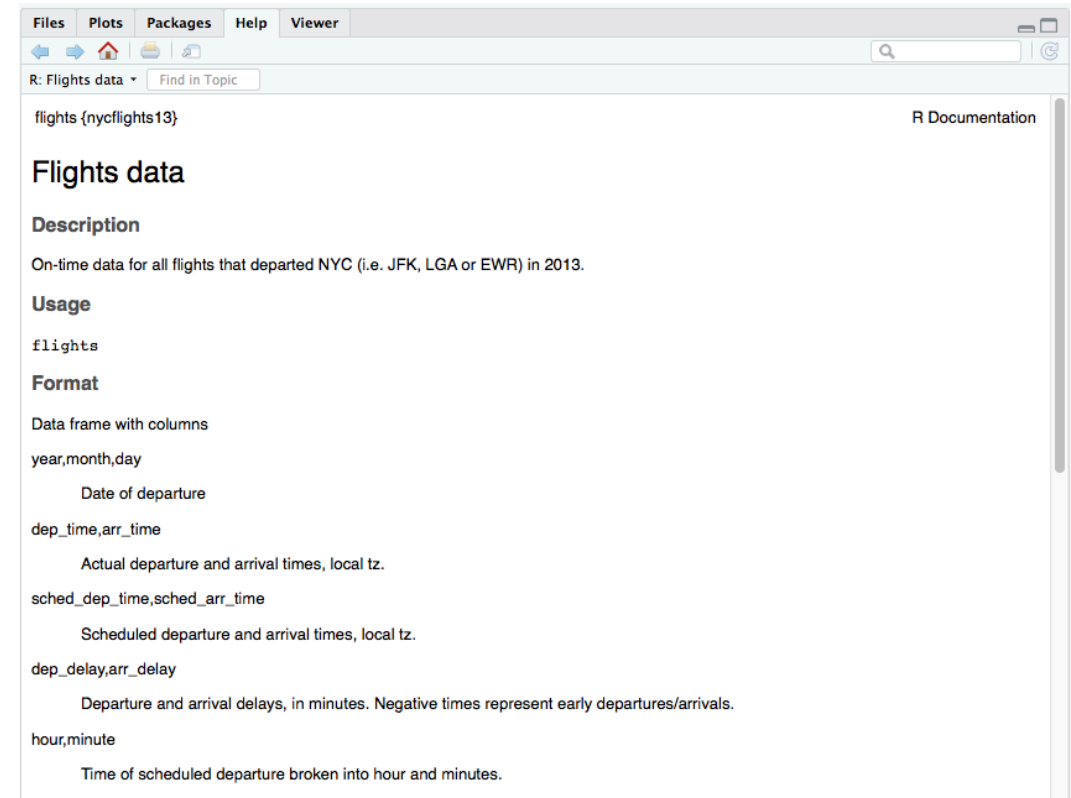
Data transformation

Let's look at the dataset we will be working with from `nycflights13` - **flights**

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

You can find the documentation for this dataset like this:

```
?flights
```



The screenshot shows the R documentation window for the 'flights' dataset. The window has a menu bar with 'Files', 'Plots', 'Packages', 'Help', and 'Viewer'. Below the menu bar is a search bar with the text 'R: Flights data' and a 'Find in Topic' button. The main content area is titled 'flights {nycflights13}' and 'R Documentation'. It includes a 'Description' section stating 'On-time data for all flights that departed NYC (i.e. JFK, LGA or EWR) in 2013.', a 'Usage' section showing 'flights', and a 'Format' section describing the data frame structure with columns: 'year, month, day' (Date of departure), 'dep_time, arr_time' (Actual departure and arrival times, local tz), 'sched_dep_time, sched_arr_time' (Scheduled departure and arrival times, local tz), 'dep_delay, arr_delay' (Departure and arrival delays, in minutes. Negative times represent early departures/arrivals), and 'hour, minute' (Time of scheduled departure broken into hour and minutes).

Basics of dplyr

After getting familiar with our dataset, let's get back to the package we use - `dplyr`

There are six functions that provide verbs for the language of data manipulation - these functions will make your life as a data scientist much easier

Uses cases for these six key `dplyr` functions are listed in the table below:

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

Framework of dplyr

The framework of `dplyr` is as follows:

- 1. The first argument is a dataframe
- 2. The next arguments describe what to do with the dataframe, using the six key `dplyr` functions
- 3. The final result is a new, transformed dataframe

We will now discuss how each of these six verbs work

Knowledge Check 1



Exercise 1



Module completion checklist

Topic	Complete
Demonstrate installing a package and loading a library	✓
Define the six functions that provide verbs for the language of data manipulation, from the package dplyr	✓
Apply the filter function to subset data	
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	
Summarize columns using the summary and group by functions	
Convert wide to long data using tidyr package	
Discuss 3 reasons of why it is important to transform and wrangle data before moving forward with analyses	
Manipulate columns by using the separate and unite functions	

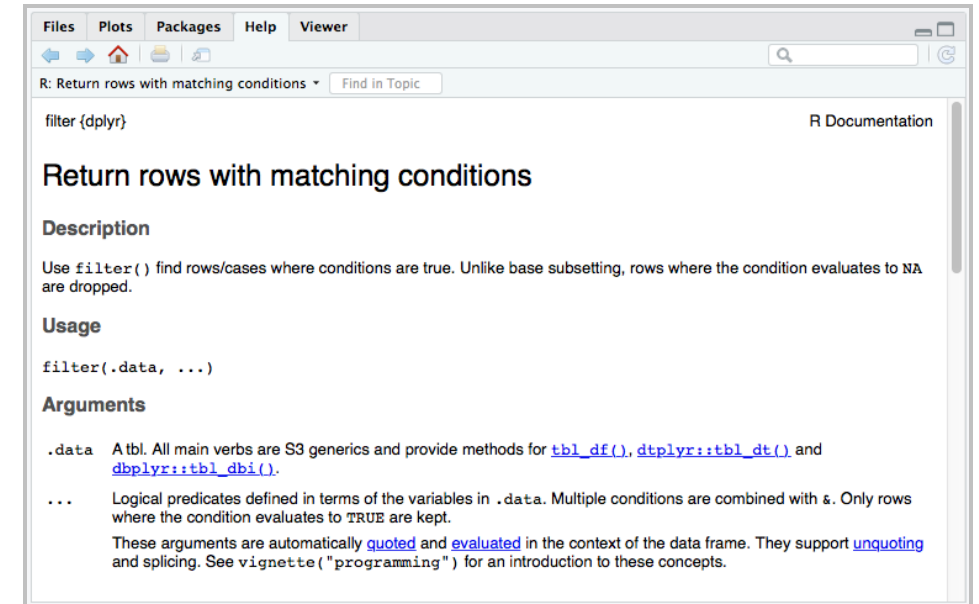
Filter

`filter` allows you to subset observations based on their values
Basic use cases for `filter` function include:

```
# Check for detailed documentation
?dplyr::filter

# Use cases for `filter` function
filter(df,           #<- dataframe
       filter_cond1, #<- subsetting rule(s)
       ...)          #<- other arguments
```

Next, we will apply `filter` on our `flights` dataset



Filter

Let's say you would like to see all flights from January 2013

```
# Load the flights dataset into the environment.
data(flights)

# Filter `flights` data frame to display all records from January (month == 1) of 2013 (year == 2013).
filter(flights,      #<- set data
        month == 1,   #<- filter by month
        year == 2013) #<- filter by year
```

```
# A tibble: 27,004 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 26,994 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 <dtm>
```

Filter

If you want to build on top of the filtered dataset, you will need to save your new subset to a new variable and perform further operations on this new subset

```
# You will have to make sure to save the subset. To do this, use `=`.  
filter_flights = filter(flights, month == 1, day == 25)  
  
# View your output.  
filter_flights
```

```
# A tibble: 922 x 19  
  year month   day dep_time sched_dep_time dep_delay arr_time  
  <int> <int> <int>   <int>         <int>      <dbl>   <int>  
1  2013     1    25      15         1815        360     208  
2  2013     1    25      17         2249         88     119  
3  2013     1    25      26         1850        336     225  
4  2013     1    25     123         2000        323     229  
5  2013     1    25     123         2029        294     215  
6  2013     1    25     456          500         -4     632  
7  2013     1    25     519          525         -6     804  
8  2013     1    25     527          530         -3     820  
9  2013     1    25     535          540         -5     826  
10 2013     1    25     539          540         -1    1006  
# ... with 912 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 <dtm>
```

Filter options

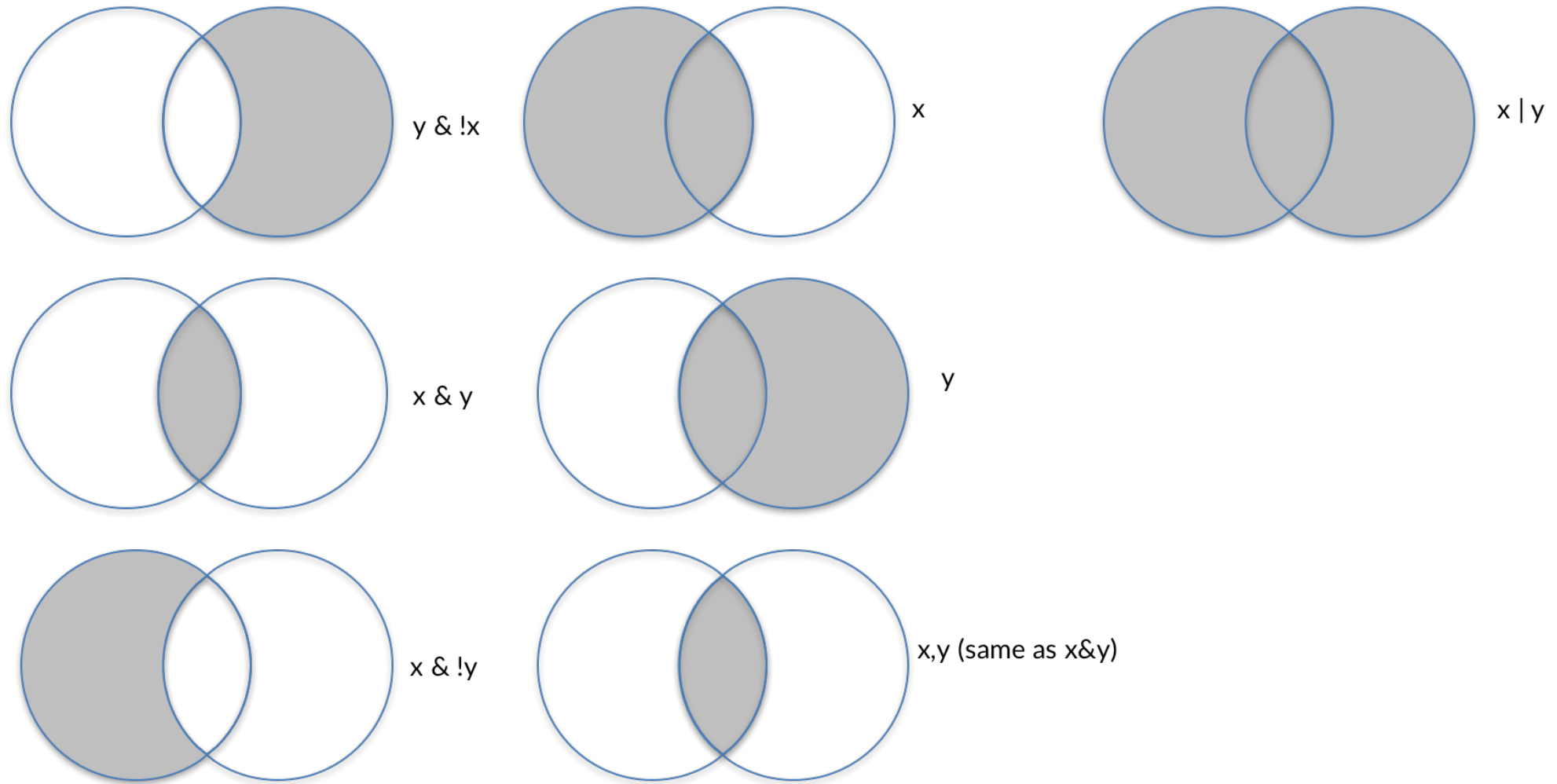
You can use the standard filtering operations when working with integer data types:

Operation	Use Case	Example
>	Greater than	6 > 4
>=	Greater than or equal to	4 >= 4
<	Less than	4 < 6
<=	Less than or equal to	4 <= 4
!=	Not equal to	4 != 6
==	Equal to	4 == 4

And more general operators:

Operation	Use Case	Example
OR or	either can be true to satisfy	x == 4 OR x == 12, x==2 x==13
and, &	and, both need to be true	x == 4 & y == 2
!	Not true, inverse selection	x != 4
%in%	value in the following list of values	x %in% c(4,16,32)

Filter - logical operators



Filter - examples of logical operators

What if we want to see all flights from January **and** on the 25th?

```
# Filter with just `&`.
filter(flights, month == 1 & day == 25)
```

```
# A tibble: 922 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>   <int>         <int>         <dbl>   <int>
1  2013     1    25      15           1815          360     208
2  2013     1    25      17           2249           88     119
3  2013     1    25      26           1850          336     225
4  2013     1    25     123           2000          323     229
5  2013     1    25     123           2029          294     215
6  2013     1    25     456            500           -4     632
7  2013     1    25     519            525           -6     804
8  2013     1    25     527            530           -3     820
9  2013     1    25     535            540           -5     826
10 2013     1    25     539            540           -1    1006
# ... with 912 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 <dtm>
```

Note: After running each example, we will record the number of rows. This will help illustrate each operator and how different a simple change of one boolean operator can have on the dataset.

Filter - examples of logical operators (cont...)

What if we want to see all flights, but **exclude** those from January and those on the 25th?

```
# Filter with `!`.
filter(flights, month != 1 & day != 25)
```

```
# A tibble: 299,597 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time
<int> <int> <int>   <int>         <int>         <dbl>   <int>
1  2013     10      1     447             500          -13     614
2  2013     10      1     522             517           5     735
3  2013     10      1     536             545          -9     809
4  2013     10      1     539             545          -6     801
5  2013     10      1     539             545          -6     917
6  2013     10      1     544             550          -6     912
7  2013     10      1     549             600         -11     653
8  2013     10      1     550             600         -10     648
9  2013     10      1     550             600         -10     649
10 2013     10      1     551             600          -9     727
# ... with 299,587 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 <dtm>
```

Here we are looking for all flights that are **not in January** and **not on the 25th**; total number of rows should be **299,587**

Filter - examples of logical operators (cont...)

```
# Filter with `%in%`.
filter(flights, month %in% c(1, 2) & day == 25)
```

```
# A tibble: 1,883 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>   <int>         <int>         <dbl>   <int>
1  2013     1    25      15          1815           360     208
2  2013     1    25      17          2249            88     119
3  2013     1    25      26          1850           336     225
4  2013     1    25     123          2000           323     229
5  2013     1    25     123          2029           294     215
6  2013     1    25     456           500            -4     632
7  2013     1    25     519           525            -6     804
8  2013     1    25     527           530            -3     820
9  2013     1    25     535           540            -5     826
10 2013     1    25     539           540            -1    1006
# ... with 1,873 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 <dtm>
```

This is a combination of `&` and `%in%` subsetting **all flights from January and February** that are **on the 25th**; number of rows should be **1,873**

Using filter with NA values

`filter` only includes rows where the condition is TRUE; it **excludes** both FALSE and NA values
If you want to preserve missing values, ask for them explicitly

```
# Create a data frame 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  
  
# Filter without specifying anything regarding NAs.  
filter(NA_df, x >= 1)
```

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

```
# Filter with specifying to keep rows if there is an NA.  
filter(NA_df, is.na(x) | x >= 1)
```

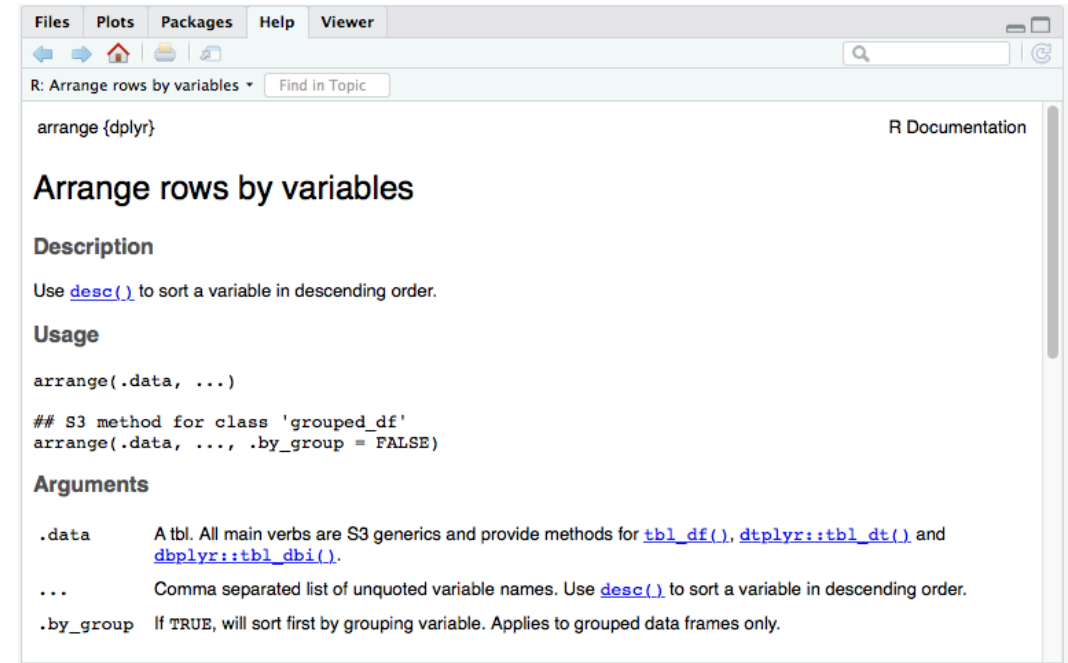
```
  x y  
1 1 1  
2 NA 2  
3 2 3
```

Arrange

arrange is used to change the order of rows within the specified column(s)
It is the equivalent of sort in SAS or order by in SQL

```
# Check for detailed documentation
?dplyr::arrange

# Use cases for `arrange` function
arrange(df,                #<- data frame
         arrange_cond1,    #<- column by which
                           #   to arrange
         ...)              #<- other args.
```



The screenshot shows the R documentation for the `arrange` function from the `dplyr` package. The window title is "R: Arrange rows by variables". The content includes the function signature `arrange {dplyr}`, a description of its purpose, usage examples, and a list of arguments with their descriptions.

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

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

Arrange example

When using multiple columns with `arrange`, the additional columns will be used to break ties in the values of preceding columns

```
# Arrange data by year, then month, and then day.
arrange(flights, #<- data frame 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 <dtm>
```

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,      #<- data frame 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 <dtm>
```

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

```
# 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
```


Knowledge Check 2



Exercise 2



Module completion checklist

Topic	Complete
Demonstrate installing a package and loading a library	✓
Define the six functions that provide verbs for the language of data manipulation, from the package dplyr	✓
Apply the filter function to subset data	✓
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	
Summarize columns using the summary and group by functions	
Convert wide to long data using tidyr package	
Discuss 3 reasons of why it is important to transform and wrangle data before moving forward with analyses	
Manipulate columns by using the separate and unite functions	

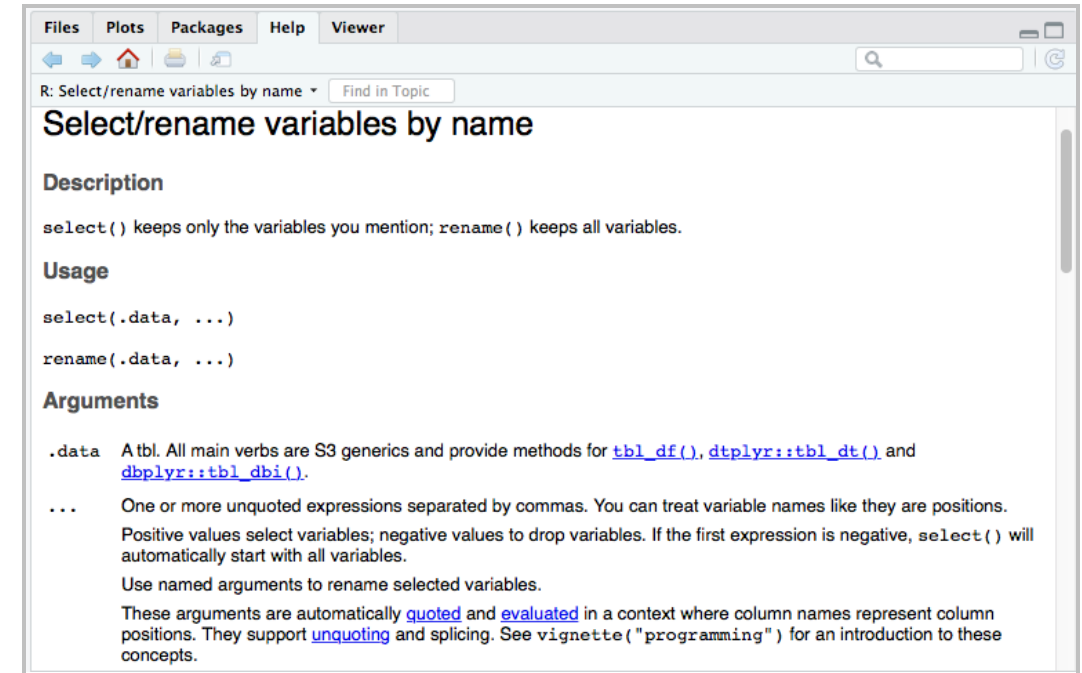
Select

`select` helps you select specific columns within your dataframe

```
# Check for detailed documentation
?dplyr::select

# Use cases for `select` function
select(df,           #<- dataframe
       select_cond1, #<- selection rule(s)
       ...)
```

We often use this function with pipes(`%>%`) which we will cover it later in this lesson
The selection criteria can be written in multiple ways, shown in the next couple of slides



Select a subset

Simply specify the column name(s)

```
# Select columns from `flights` data frame.
select(flights, #<- specify the data frame
        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` data frame
select(flights, #<- specify the data frame
        year:day) #<- specify the range of
columns
```

```
# 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 exclusion operator (i.e. -)

```
# Select multiple columns from `flights` data frame by providing which columns to exclude in selection
select(flights,          #<- specify the data frame
       -(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 <dtm>
```

Select - helper functions

Helpers are multiple functions you can use to select variables based on their names

They act like regular expressions but in a more simplified manner

Here are some of the more commonly used helper functions:

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

To select columns whose names start with 'arr':

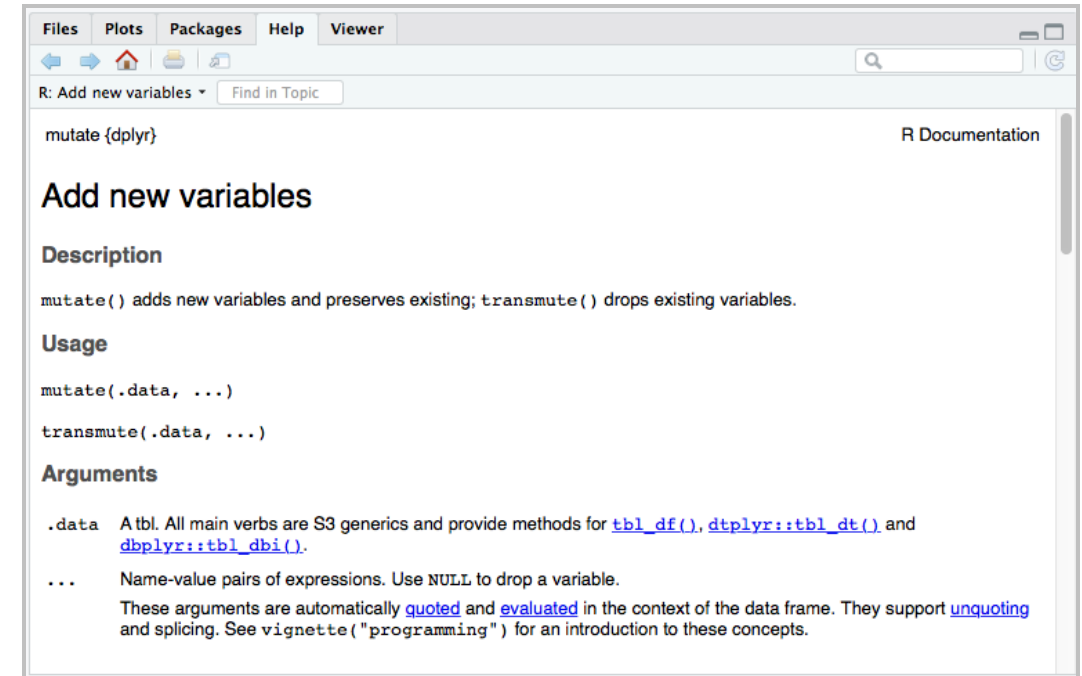
```
select(flights, starts_with("arr"))
```

Mutate

`mutate` is an essential function of `dplyr`
It allows us to **create** new variables using
the current data and **append** these
variables to the existing dataframe

```
?dplyr::mutate  
  
mutate(df,      # <- dataframe  
       new_col1, # <- rule(s) for the new column  
       ...)
```

Mutate always adds columns to the end of
the dataset, so make sure you are able to
see the last columns



The screenshot shows the R Documentation page for the `mutate` function from the `dplyr` package. The page is titled "mutate {dplyr}" and includes a search bar at the top. The main content is organized into sections: "Add new variables", "Description", "Usage", and "Arguments". The "Description" section states that `mutate()` adds new variables and preserves existing ones, while `transmute()` drops existing variables. The "Usage" section shows the function signature `mutate(.data, ...)` and `transmute(.data, ...)`. The "Arguments" section explains that `.data` is a tibble and `...` represents name-value pairs of expressions. It also mentions that arguments are automatically quoted and evaluated in the context of the data frame.

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 tibble. 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.

Mutate

Create the dataset using select

```
# Let's select columns of `flights` data frame and save them as `flights_sml`.
flights_sml = select(flights, #<- specify data frame
                      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
```

Mutate

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

```
# Add two columns `gain` and `speed` to `flights_sml`.
mutate(flights_sml,                                #<- specify the data frame
      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
```

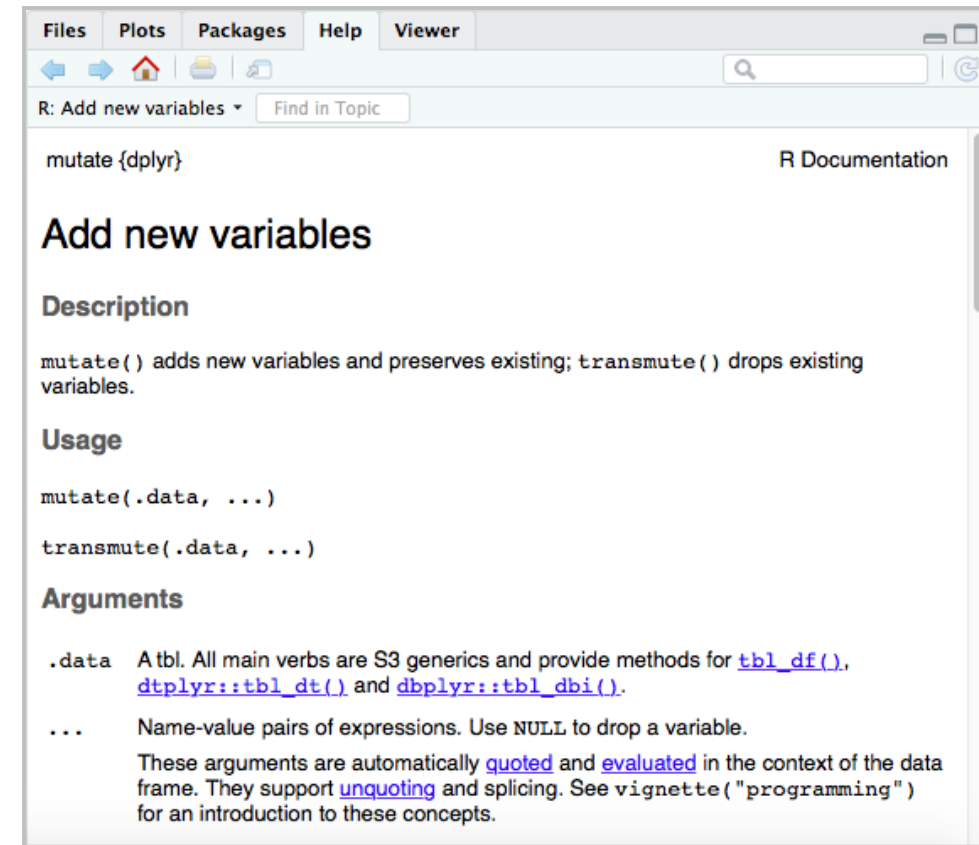
Transmute

transmute is a function that does the same thing as mutate **except it will only keep the new columns**

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

The 1st argument is the dataframe
The following arguments are the columns that will be included in your new data frame

Note: you are isolating only these new columns



The screenshot shows the RStudio interface with the 'Viewer' pane displaying the documentation for the `transmute` function. The documentation is titled 'Add new variables' and includes a description, usage, and arguments section. The description states that `mutate()` adds new variables and preserves existing ones, while `transmute()` drops existing variables. The usage section shows the function signatures: `mutate(.data, ...)` and `transmute(.data, ...)`. The arguments section explains that `.data` is a tibble and all main verbs are S3 generics, providing methods for `tbl_df()`, `dtplyr::tbl_dt()`, and `dbplyr::tbl_dbi()`. It also notes that the remaining arguments are name-value pairs of expressions, which are automatically quoted and evaluated in the context of the data frame, supporting unquoting and splicing.

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 tibble. 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

With the same arguments as in the `mutate` example, we can see `transmute` function only returns new columns

```
# Add two columns `gain` and `speed` to `flights_sml`.
example = transmute(flights_sml,                               #<- specify the data frame
  gain = arr_delay - dep_delay,                                #<- create `gain` column by subtracting departure 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
```

Mutate and transmute- useful functions

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

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

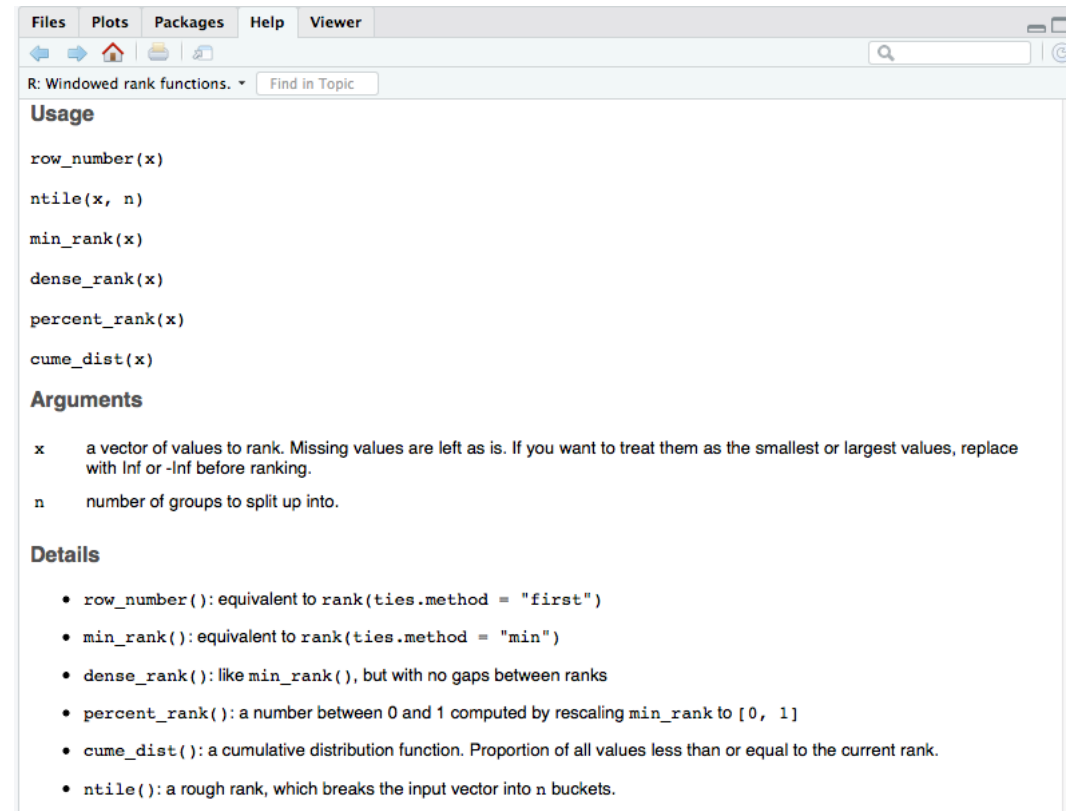
Mutate and transmute - useful functions (cont...)

Ranking functions are very helpful in data manipulation

There are several within the `dplyr` package such as `row_number()`, `ntile()` and `dense_rank()`

```
# Check for detailed documentation
?dplyr::ranking

rank_function(x) # <- one of rank functions with
                 #      a vector of values to rank
```



The screenshot shows the R help window for the `ranking` functions in the `dplyr` package. The window title is "R: Windowed rank functions." and it includes a search bar. The content is organized into sections: **Usage**, **Arguments**, and **Details**.

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.

Knowledge Check 3



Exercise 3



Module completion checklist

Topic	Complete
Demonstrate installing a package and loading a library	✓
Define the six functions that provide verbs for the language of data manipulation, from the package dplyr	✓
Apply the filter function to subset data	✓
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	✓
Summarize columns using the summary and group by functions	
Convert wide to long data using tidyr package	
Discuss 3 reasons of why it is important to transform and wrangle data before moving forward with analyses	
Manipulate columns by using the separate and unite functions	

Summarise and group_by

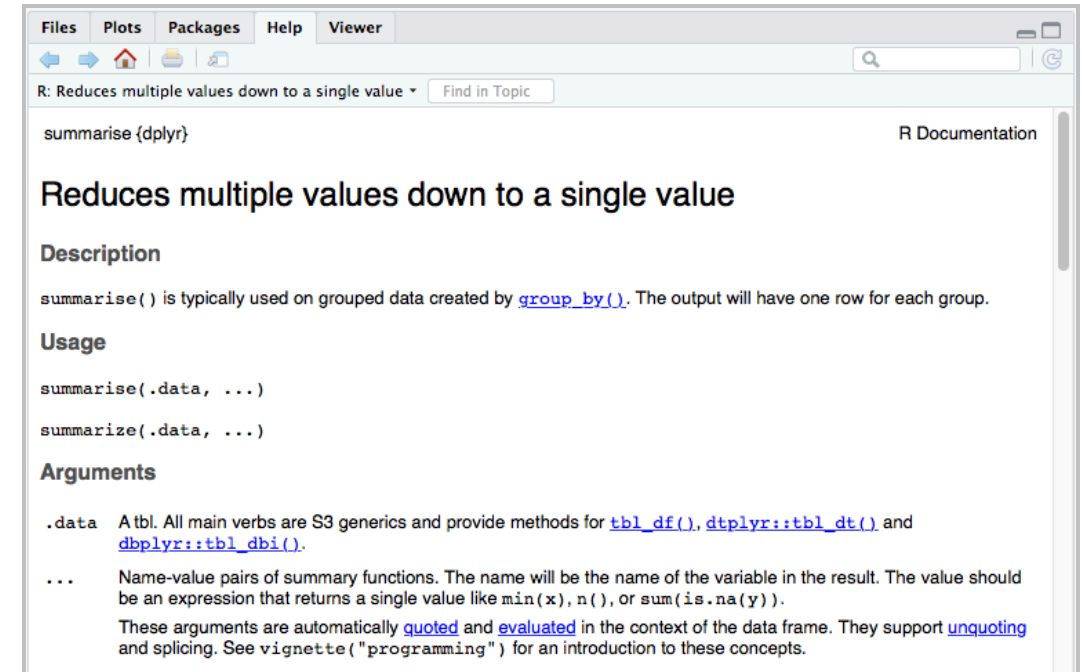
`summarise` collapses a data frame down to a single row

By itself, `summarise` is not very helpful

We will often use it with `group_by`

```
# Check for detailed documentation
?dplyr::summarise

# Use cases for `summarise` function
summarise(df,                #<- data frame
           summary_function1, #<- summary rule(s)
           ...               #   for new column
           ...)
```

A screenshot of the R Documentation window for the `summarise` function from the `dplyr` package. The window has a menu bar with 'Files', 'Plots', 'Packages', 'Help', and 'Viewer'. Below the menu bar is a search bar and a 'Find in Topic' button. The main content area is titled 'summarise (dplyr)' and 'R Documentation'. It includes a description: 'Reduces multiple values down to a single value'. Under 'Description', it says: 'summarise() is typically used on grouped data created by `group_by()`. The output will have one row for each group.' Under 'Usage', it shows two function signatures: `summarise(.data, ...)` and `summarize(.data, ...)`. Under 'Arguments', it lists:

- `.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 summary functions. The name will be the name of the variable in the result. The value should be an expression that returns a single value like `min(x)`, `n()`, or `sum(is.na(y))`. 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.

Summarise and group_by

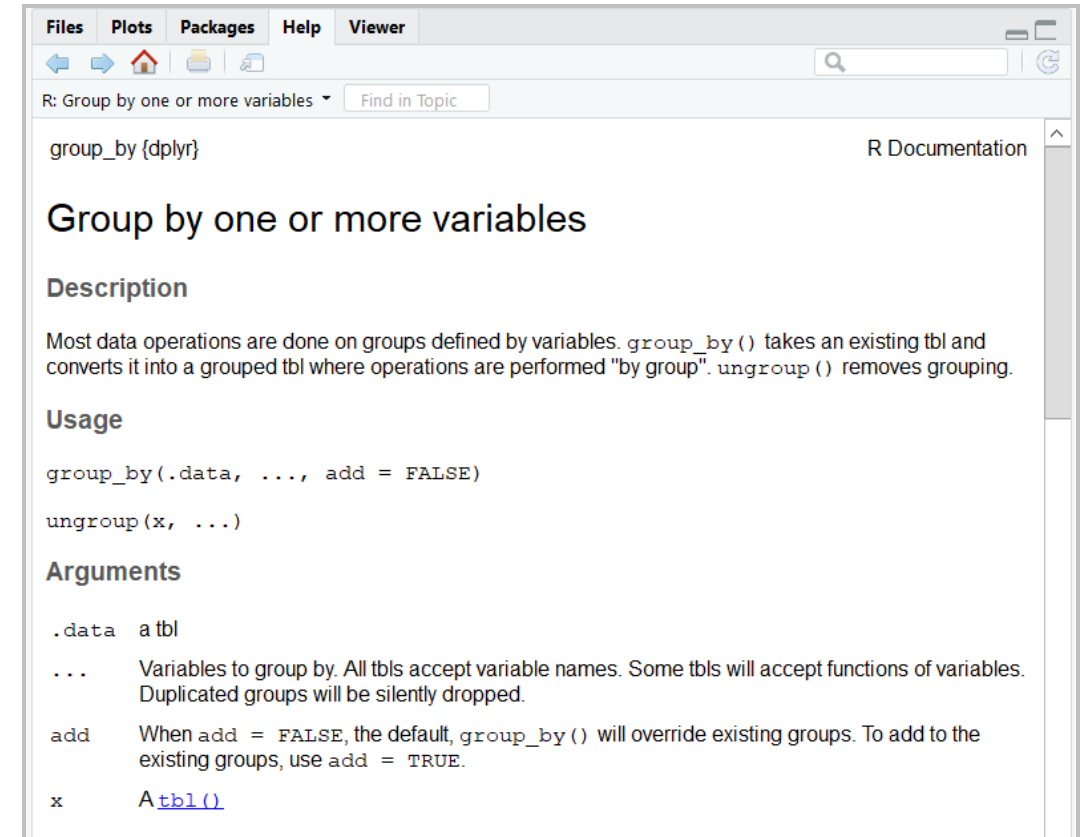
Grouping doesn't change how data looks
apart from listing how it's grouped

It will change how it acts with the other
dplyr verbs

To remove grouping, use ungroup

```
# Check for detailed documentation
?dplyr::group_by

# Use cases for `group by` function
group_by(df,           #<- data frame
         variable1,    #<- 1st variable to group
         by
         variable2,    #<- 2nd variable to group
         by
         ...)
```

A screenshot of the R Documentation window for the 'group_by' function from the dplyr package. The window has a menu bar with 'Files', 'Plots', 'Packages', 'Help', and 'Viewer'. Below the menu bar is a search bar and a dropdown menu showing 'R: Group by one or more variables'. The main content area is titled 'group_by {dplyr}' and 'R Documentation'. It includes a 'Description' section stating that most data operations are done on groups defined by variables, and that 'group_by()' takes an existing tbl and converts it into a grouped tbl. It also includes a 'Usage' section with the functions 'group_by(.data, ..., add = FALSE)' and 'ungroup(x, ...)'. Finally, it has an 'Arguments' section listing '.data' as a tbl, '...' as variables to group by, 'add' as a logical value to control whether to add to existing groups, and 'x' as a tbl object.

Files Plots Packages Help Viewer

R: Group by one or more variables Find in Topic

group_by {dplyr} R Documentation

Group by one or more variables

Description

Most data operations are done on groups defined by variables. `group_by()` takes an existing `tbl` and converts it into a grouped `tbl` where operations are performed "by group". `ungroup()` removes grouping.

Usage

```
group_by(.data, ..., add = FALSE)
ungroup(x, ...)
```

Arguments

<code>.data</code>	a <code>tbl</code>
<code>...</code>	Variables to group by. All <code>tbls</code> accept variable names. Some <code>tbls</code> will accept functions of variables. Duplicated groups will be silently dropped.
<code>add</code>	When <code>add = FALSE</code> , the default, <code>group_by()</code> will override existing groups. To add to the existing groups, use <code>add = TRUE</code> .
<code>x</code>	A <code>tbl()</code> .

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>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>,
#   minute <dbl>, time_hour <dtm>
```

Summarise and group_by together

```
# 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 continue to 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 %>%  
  group_by(dest) %>%  
  summarise(count = n(),  
            dist = mean(distance, na.rm = TRUE),  
            delay = mean(arr_delay, na.rm = TRUE)) %>%  
  filter(count > 20, dest != "HNL")  
delays
```

#<- take flights data
#<- group it by destination
#<- then summarize by creating count variable
#<- and computing mean distance
#<- and mean arrival delay
#<- then filter it

```
# 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
```

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

Apart from `mean()`, there are many other summary functions describing data from various aspects:

Summary Functions	Explanation
<code>n()</code>	will count the number of entries that come from a summarise
<code>min(x), quantile(x, 0.25), max(x)</code>	measures of rank and distribution can be used
<code>first(x), nth(x, 2), last(x)</code>	measures of position and order
<code>n_distinct</code>	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
```

Summarise not needed to count

count is a simple count function that does not require the 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> <int> <int>  
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> <int> <int>
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 data frame
  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
```

Data wrangling

Data transformation is where you get the dataset ready for wrangling

We want all the variables and values, all the new columns to be created, and all the NAs taken care of before ensuring it is in `tidy` form

`tidyr`, the package within `tidyverse`, allows us to get our data into a tidy format

We will use the `.Rdata` file loaded at the beginning of this lesson to demonstrate

For further reading and understanding of tidy data and where it originated, check out this [paper](#)

Would analysis be easy with these datasets?

Here is a list of objects from our RData file

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'?

These three interrelated rules make a dataset tidy:

- 1. Each variable must have its own column
- 2. Each observation must have its own row
- 3. 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

Storing data in a **consistent** way:

- 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**:

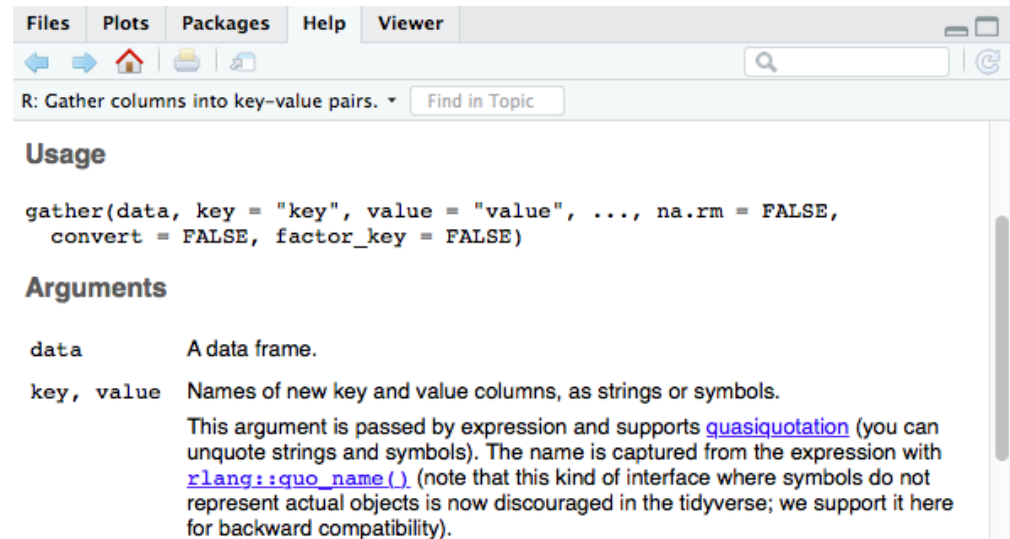
- The functions of `tidyr` that will help you transform `messy` data to `tidy` data

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`:

- 1. The set of columns that represent the values
- 2. The name of the variable (which we decide upon) that represents those values, or the `key`
- 3. The name of the variable (which we decide upon) that represents the values that are currently within the value columns, or the `value`

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 combined into one variable, `year`

Let's use `gather` to combine 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` data frame to make it tidy.
year_country %>% #<- set the data frame 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 the packages within `tidyverse`!

Gather function: specifying a range

```
# Gather the `year_country` data frame to make it tidy.
year_country %>% #<- set the data frame 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` with the named columns

Spreading

spread spreads one column into multiple variables

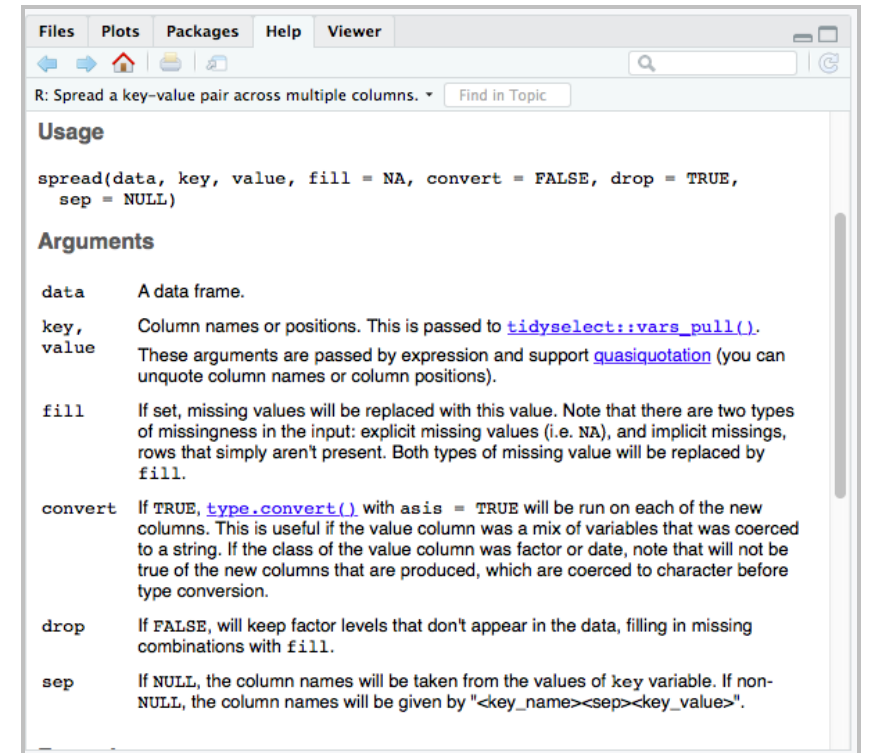
Spreading is the opposite of gathering

You use spread when an observation is scattered across multiple rows

```
?dplyr::spread  
  
summarise(df,      #<- data frame  
          key,      #<- name of current key column  
          value)    #<- name of current value column
```

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 data frame

Use `spread` with **2 main parameters**:

- 1. The `key`, which contains the variables
- 2. 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.
# Data frame 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
```

Separating and uniting

But how would we adjust a single variable?

What would we use for a data frame 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 complement of `separate` is `unite`

We will learn how to use this as well

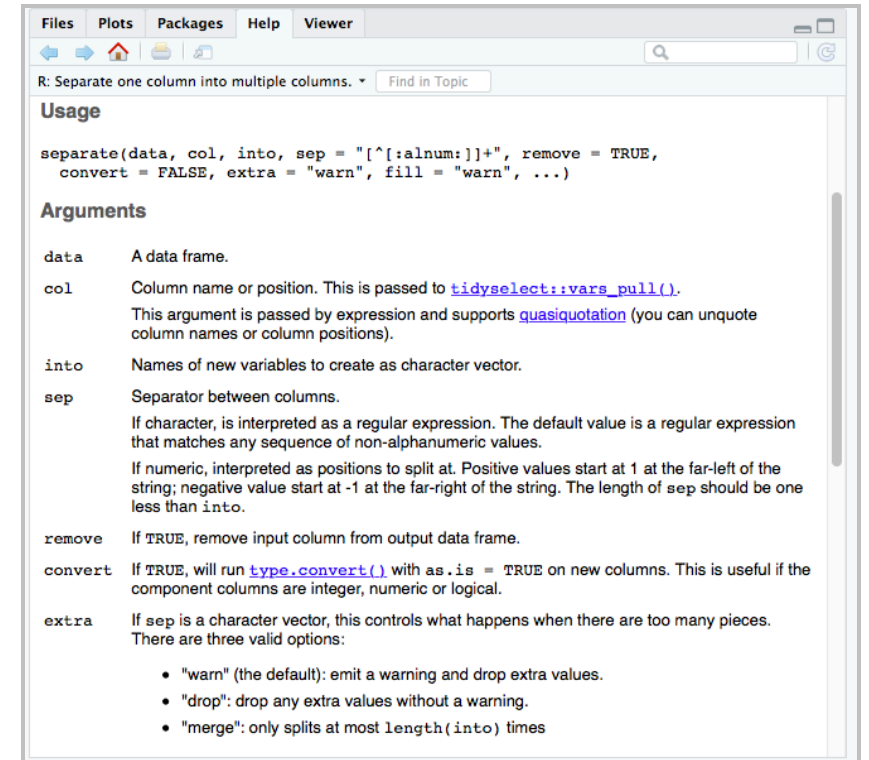
Separate

`separate` separates a single character column into multiple columns and takes two arguments:

- The first argument is the dataframe
- Next we pipe it to `separate`
- The first parameter is the column to be separated
- The second parameter defines how we want to separate the variable, using `into = c("var_1", "var_2")`

```
?dplyr::separate
```

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



The screenshot shows the R help documentation for the `separate` function. The title bar indicates the topic is "R: Separate one column into multiple columns." The "Usage" section shows the function signature: `separate(data, col, into, sep = "[^[:alnum:]]+", remove = TRUE, convert = FALSE, extra = "warn", fill = "warn", ...)`. The "Arguments" section lists the parameters: `data` (A data frame), `col` (Column name or position), `into` (Names of new variables), `sep` (Separator between columns), `remove` (If TRUE, remove input column), `convert` (If TRUE, run `type.convert()`), and `extra` (Controls what happens when there are too many pieces). The `extra` argument has three options: "warn" (default), "drop", and "merge".

Separate

```
# Using `rate_country` separate its `rate` column into two.
rate_country %>%                                     #<- set data frame 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 by which to separate

```
# Using `rate_country` separate its `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 its `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

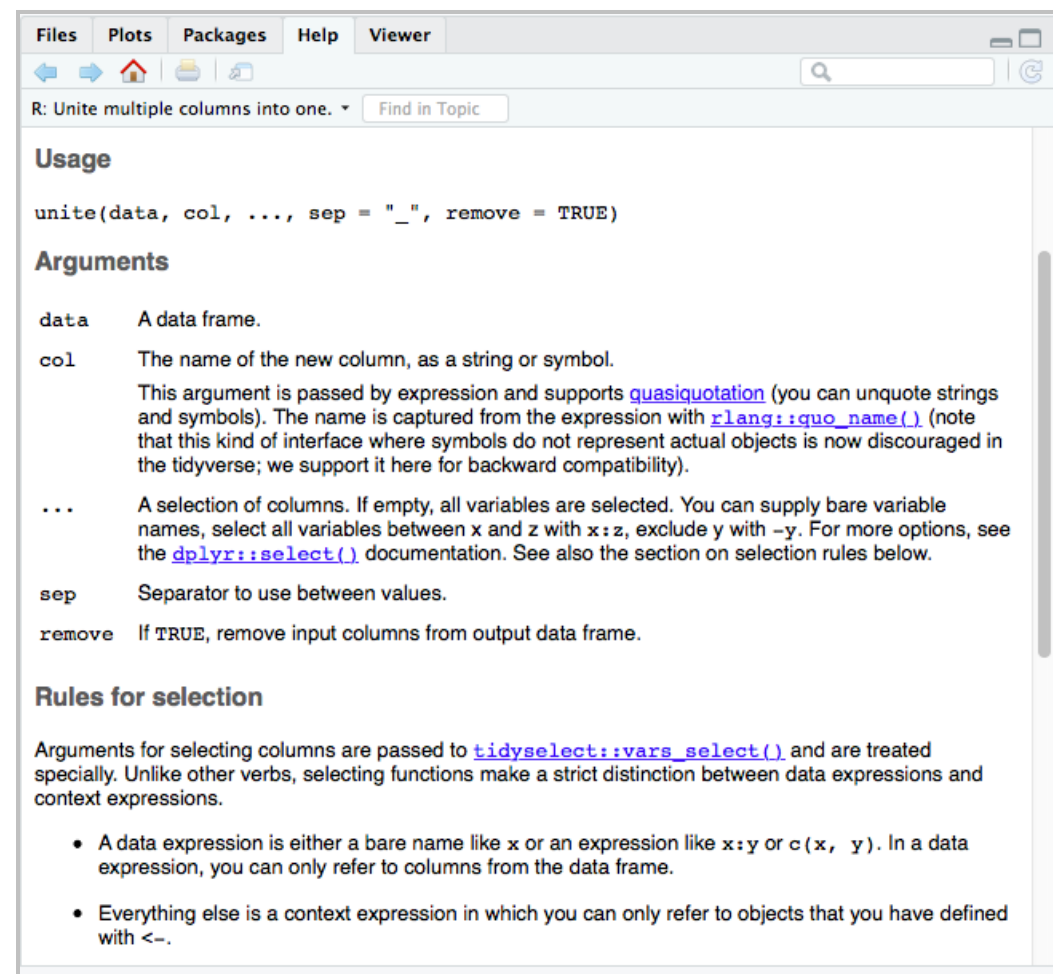
```
rate_country %>%  
  separate(rate, into = c("cases", "population"), convert =  
    TRUE)
```

```
# 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::separate  
  
unite(df, #<- data frame  
      col, #<- name of column to unite  
      sep) #<- separator to use
```

unite combines multiple character columns into a single column
unite is the inverse of separate



The screenshot shows an R help window for the `unite` function. The window has a menu bar with 'Files', 'Plots', 'Packages', 'Help', and 'Viewer'. Below the menu bar is a search bar and a 'Find in Topic' button. The main content area is titled 'R: Unite multiple columns into one.' and contains the following information:

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 <code>x:z</code> , exclude y with <code>-y</code> . 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 %>%
  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 4



Exercise 4



Module completion checklist

Topic	Complete
Demonstrate installing a package and loading a library	✓
Define the six functions that provide verbs for the language of data manipulation, from the package dplyr	✓
Apply the filter function to subset data	✓
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	✓
Summarize columns using the summary and group by functions	✓
Convert wide to long data using tidyr package	✓
Discuss 3 reasons of why it is important to transform and wrangle data before moving forward with analyses	✓
Manipulate columns by using the separate and unite functions	✓

Review!



Summary

Topics		
Week 1-2	Intro to R programming	
Week 3-5	Machine Learning - Regression and Unsupervised Learning	
Week 6-8	Machine Learning - Classification	

In today's module, we learn two powerful packages in `tidyverse`: **dplyr** and **tidyr**

With these two packages, we can manipulate data and tidy up datasets in a more elegant way

After class, you can perform your own transformation with other built-in R datasets

In the next module, we will **visualize** our data and make our analysis results more accessible.

Stay excited!

This completes our module
Congratulations!