DATA SOCIETY®

Intro to R - Part 3

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

Welcome back!

- In the previous lessons, we discussed:
  - Naming variables in R
  - Distinguishing between different data types
  - Basic data structures like atomic vectors, matrices, lists, dataframes and performed operations around them
- In this module, we will learn how to use control flow structures in R, perform data cleaning operations and manipulate data using various wrangling packages like `tidyverse` and `dplyr`

Warm up

Before we start, let's all check out part of this video to learn how Airbnb uses R to work more effectively with data:

https://www.youtube.com/watch?v=70luTZU-D3E&ab_channel=Work-Bench

- **We will only watch from 1:00 mark-to 4:10 today**, but feel free to watch the rest on your own time!

Module completion checklist

Objective	Complete
Load and evaluate the dataset	
Address missing values in data	
Manipulate data types and structures using flow control structures (`for` loops, conditionals,etc)	
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	

Introducing CMP dataset

- We are going to explore a new dataset called `ChemicalManufacturingProcess` from `AppliedPredictiveModeling` package in R
- This dataset contains information about a chemical manufacturing **process**
- The goal is to understand the relationship between the process and the resulting **yield**
- Raw material in this process is put through a sequence of 27 steps to generate the final pharmaceutical product
- Of the 57 characteristics, there are:
  - **12 measurements of** the biological starting **material**, and
  - **45 measurements of** the manufacturing **process**
- The starting **material** is generated from a biological unit and has a range of quality and characteristics
- The **process** variables include measurements such as temperature, drying time, washing time, and concentrations of byproducts at various steps

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 `skillsoft-2021` folder

```
# Set `main_dir` to the location of your `skillsoft-2021` folder (for Mac/Linux).
main_dir = "~/Desktop/skillsoft-2021"

# Set `main_dir` to the location of your `skillsoft-2021` folder (for Windows).
main_dir = "C:/Users/[username]/Desktop/skillsoft-2021"

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

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

Directory settings

- Now all you have to do to switch between working directories is to use a variable instead of typing the full path every time

```
# Set working directory to where the data is.
setwd(data_dir)
```

```
# Print working directory (Mac/Linux).
getwd()
```

```
[1] "/home/[your-user-name]/Desktop/skillsoft-2021/data"
```

```
# Print working directory (Windows).
getwd()
```

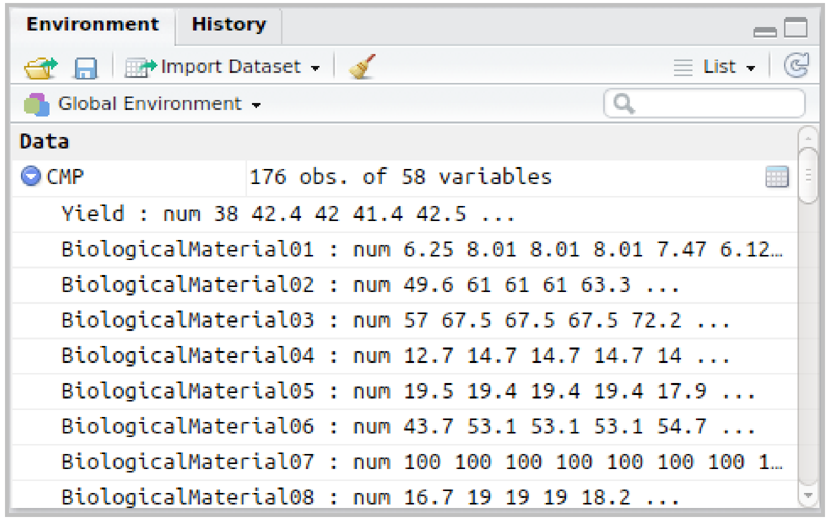
```
[1] "C:/Users/[your-user-name]/Desktop/skillsoft-2021/data"
```

Loading the dataset

- Let's load the dataset from our `data_dir` into R's environment

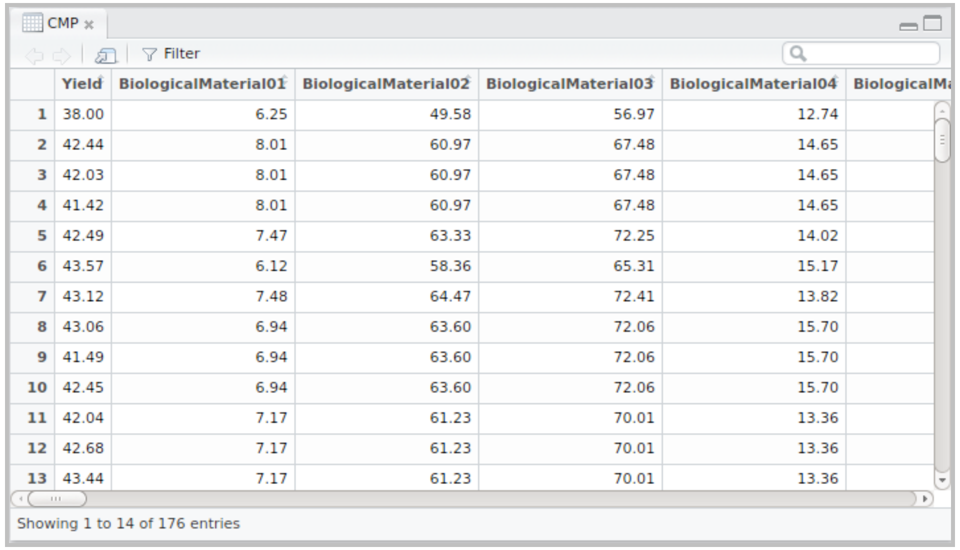
```r
# Set working directory to where we store data.
setwd(data_dir)

# Read CSV file called
"ChemicalManufacturingProcess.csv"
CMP = read.csv("ChemicalManufacturingProcess.csv",
               header = TRUE,
               stringsAsFactors = FALSE)
```



- The dataset consists of `176` observations and `58` variables

```r
# View CMP dataset in tabular data
explorer.
View(CMP)
```



	Yield	BiologicalMaterial01	BiologicalMaterial02	BiologicalMaterial03	BiologicalMaterial04	BiologicalMa
1	38.00	6.25	49.58	56.97	12.74	
2	42.44	8.01	60.97	67.48	14.65	
3	42.03	8.01	60.97	67.48	14.65	
4	41.42	8.01	60.97	67.48	14.65	
5	42.49	7.47	63.33	72.25	14.02	
6	43.57	6.12	58.36	65.31	15.17	
7	43.12	7.48	64.47	72.41	13.82	
8	43.06	6.94	63.60	72.06	15.70	
9	41.49	6.94	63.60	72.06	15.70	
10	42.45	6.94	63.60	72.06	15.70	
11	42.04	7.17	61.23	70.01	13.36	
12	42.68	7.17	61.23	70.01	13.36	
13	43.44	7.17	61.23	70.01	13.36	

Showing 1 to 14 of 176 entries

Subsetting data

- In this module, we will explore a subset of this dataset, which includes the following variables
  - **yield**
  - **3 material** variables, and
  - **3 process** variables

CMP

Filter

	Yield	BiologicalMaterial01	BiologicalMaterial02	BiologicalMaterial03
1	38.00	6.25	49.58	56.97
2	42.44	8.01	60.97	67.48
3	42.03	8.01	60.97	67.48
4	41.42	8.01	60.97	67.48
5	42.49	7.47	63.33	72.25
6	43.57	6.12	58.36	65.31
7	43.12	7.48	64.47	72.41
8	43.06	6.94	63.60	72.06
9	41.49	6.94	63.60	72.06

Showing 1 to 10 of 176 entries

...

CMP

Filter

ManufacturingProcess01	ManufacturingProcess02	ManufacturingProcess03
NA	NA	NA
0.0	0.0	NA
0.0	0.0	NA
0.0	0.0	NA
10.7	0.0	NA
12.0	0.0	NA
11.5	0.0	1.56
12.0	0.0	1.55
12.0	0.0	1.56

Showing 1 to 10 of 176 entries

Subsetting data

```
# Let's make a vector of column indices we would like to save.
column_ids = c(1:4,   #<- concatenate a range of ids
               14:16)#<- with another a range of ids
column_ids            #<- verify that we have the correct set of columns
```

```
[1]  1  2  3  4 14 15 16
```

```
# Let's save the subset into a new variable and look at its structure.
CMP_subset = CMP[ , column_ids]
str(CMP_subset)
```

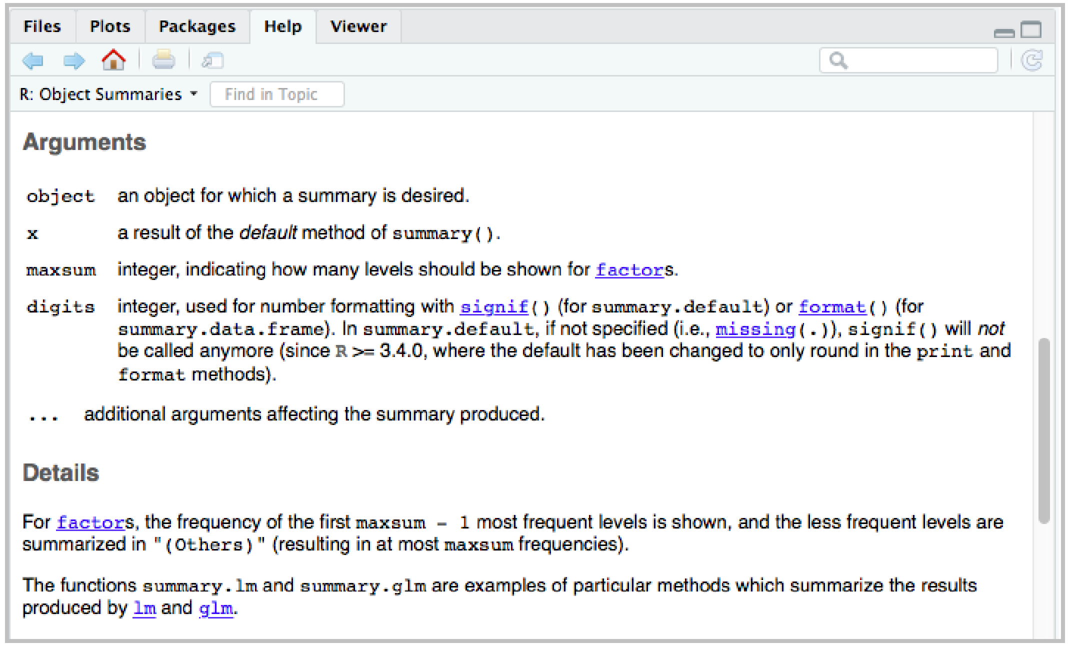
```
'data.frame':	176 obs. of  7 variables:
 $ Yield                 : num  38 42.4 42 41.4 42.5 ...
 $ BiologicalMaterial01  : num  6.25 8.01 8.01 8.01 7.47 6.12 7.48 6.94 6.94 6.94 ...
 $ BiologicalMaterial02  : num  49.6 61 61 61 63.3 ...
 $ BiologicalMaterial03  : num  57 67.5 67.5 67.5 72.2 ...
 $ ManufacturingProcess01: num  NA 0 0 0 10.7 12 11.5 12 12 12 ...
 $ ManufacturingProcess02: num  NA 0 0 0 0 0 0 0 0 0 ...
 $ ManufacturingProcess03: num  NA NA NA NA NA NA 1.56 1.55 1.56 1.55 ...
```

Summary statistics

- To get quick summary statistics of your dataframe, or one single column within the dataframe, use `summary`

```
?summary

summary(data) #<- Either the dataframe or single column
```



Summary statistics: CMP

```
summary(CMP_subset) #<- getting summary statistics of CMP_subset
```

```
     Yield       BiologicalMaterial01 BiologicalMaterial02 BiologicalMaterial03
 Min.   :35.25   Min.   :4.580        Min.   :46.87        Min.   :56.97
 1st Qu.:38.75   1st Qu.:5.978        1st Qu.:52.68        1st Qu.:64.98
 Median :39.97   Median :6.305        Median :55.09        Median :67.22
 Mean   :40.18   Mean   :6.411        Mean   :55.69        Mean   :67.70
 3rd Qu.:41.48   3rd Qu.:6.870        3rd Qu.:58.74        3rd Qu.:70.43
 Max.   :46.34   Max.   :8.810        Max.   :64.75        Max.   :78.25

 ManufacturingProcess01 ManufacturingProcess02 ManufacturingProcess03
 Min.   : 0.00          Min.   : 0.00          Min.   :1.47
 1st Qu.:10.80          1st Qu.:19.30          1st Qu.:1.53
 Median :11.40          Median :21.00          Median :1.54
 Mean   :11.21          Mean   :16.68          Mean   :1.54
 3rd Qu.:12.15          3rd Qu.:21.50          3rd Qu.:1.55
 Max.   :14.10          Max.   :22.50          Max.   :1.60
 NA's   :1              NA's   :3              NA's   :15
```

Module completion checklist

Objective	Complete
Load and evaluate the dataset	✔
Address missing values in data	
Manipulate data types and structures using flow control structures (`for` loops, conditionals,etc)	
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	

Working with missing data: max values

```
# Let's try and compute the maximum value of the 1st manufacturing process.
max_process01 = max(CMP_subset$ManufacturingProcess01)
max_process01
```

```
[1] NA
```

- Notice that we get `NA` in return

```
max_process02 = max(CMP_subset$ManufacturingProcess01, na.rm = TRUE)
max_process02
```

```
[1] 14.1
```

- We now get an actual number by using `na.rm = TRUE` to ignore NA values

Working with missing data: imputing

- What if the function you are using does not have the method `na.rm`? Or what if removing `NA`s skews the results?
- Data imputation with one of the following values will help to overcome this:
  - `0`
  - `mean`
  - `median`
  - any other special value appropriate for a given dataset and data type (e.g. handling of categorical variables with missing data should be handled differently from imputing numeric variables)
- Replacing `NA`s with **`mean`** may not work well if the data contains outliers

Working with missing data

- Function `is.na` will provide a vector of `TRUE` or `FALSE` for each element of a given vector
- It is hard to track elements that are indeed `NA` for datasets containing even a moderate number of data points

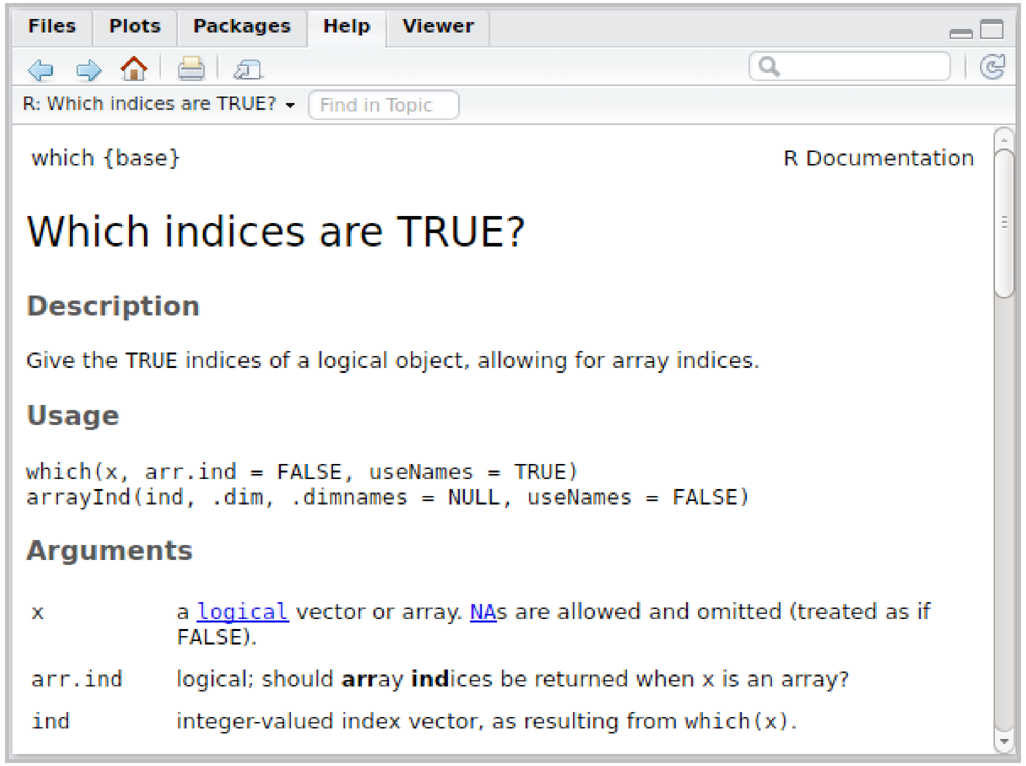
```
# Let's take a look at `ManufacturingProcess01`
# and see if any of the values in it are `NA`.
is.na(CMP_subset$ManufacturingProcess01)
```

```
  [1]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [25] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [49] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [61] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [73] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [85] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [97] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[109] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[121] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[133] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[145] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[157] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[169] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

Working with missing data

```
?which
```

- `which` function is an invaluable utility function in R's `base` package
- It takes either a vector/array of `logical` values OR a vector/array of any values with a comparison statement in one of the comparison operators (==, !=, >, <, >=, <=) and a value to which we are comparing to
- It returns the `indices` of all `TRUE` values of the `logical` vector, or the indices of all the values that meet the condition we specified



Files Plots Packages Help Viewer

R: Which indices are TRUE? Find in Topic

which {base} R Documentation

Which indices are TRUE?

Description

Give the TRUE indices of a logical object, allowing for array indices.

Usage

```
which(x, arr.ind = FALSE, useNames = TRUE)
arrayInd(ind, .dim, .dimnames = NULL, useNames = FALSE)
```

Arguments

x	a logical vector or array. NAs are allowed and omitted (treated as if FALSE).
arr.ind	logical; should **arr**ay **ind**ices be returned when x is an array?
ind	integer-valued index vector, as resulting from which(x).

Working with missing data: identifying NA values

```r
# Let's save this vector of logical values to a variable.
is_na_MP01 = is.na(CMP_subset$ManufacturingProcess01)

# To determine WHICH elements in the vector are `TRUE`and are NA, we will use `which` function.

# Since we already have a vector of `TRUE` or `FALSE` logical values
# we only have to give it to `which` and it will return all of the
# indices of values that are `TRUE`.
which(is_na_MP01)
```

```
[1] 1
```

```r
# This is also a correct way to set it up.
which(is_na_MP01 == TRUE)
```

```
[1] 1
```

Working with missing data: locating NA values

- Now that we know which entry in the `ManufacturingProcess01` is `NA`, we can select it programmatically without having to type its index manually

```
# Let's save the index to a variable.
na_id = which(is_na_MP01)
na_id
```

```
[1] 1
```

```
# Let's view the value at the `na_id` index.
CMP_subset$ManufacturingProcess01[na_id]
```

```
[1] NA
```

Working with missing data: mean replacement

- We need to compute a value suitable for replacing the given `NA`
- For demonstration purposes we will use the `mean` of the variable as a replacement

```
# Compute the mean of the `ManufacturingProcess01`.
mean_process01 = mean(CMP_subset$ManufacturingProcess01)
mean_process01
```

```
[1] NA
```

- Set `na.rm = TRUE` in order to compute the `mean` of the variable that contains `NA`s!

```
# Compute the mean of the `ManufacturingProcess01` and set `na.rm` to `TRUE`.
mean_process01 = mean(CMP_subset$ManufacturingProcess01, na.rm = TRUE)
mean_process01
```

```
[1] 11.20743
```

Working with missing data

- We can now take the mean and assign it to the missing value within the vector

```
# Assign the mean to the entry with the `NA`.
CMP_subset$ManufacturingProcess01[na_id] = mean_process01
CMP_subset$ManufacturingProcess01[na_id]
```

```
[1] 11.20743
```

- Now, instead of the `NA`, we have the `mean` value of this column!
- Let's compute the `max` of the column without `na.rm` specified to see if it works:

```
max_process01 = max(CMP_subset$ManufacturingProcess01)
max_process01
```

```
[1] 14.1
```

Working with missing data

- Next, we repeat the process for the remaining manufacturing variables

```
# Impute missing values of `ManufacturingProcess02` with the mean
is_na = is.na(CMP_subset$ManufacturingProcess02)
na_id = which(is_na)
mean_process02 = mean(CMP_subset$ManufacturingProcess02, na.rm = TRUE)
CMP_subset$ManufacturingProcess02[na_id] = mean_process02


# Impute missing values of `ManufacturingProcess03` with the mean
is_na = is.na(CMP_subset$ManufacturingProcess03)
na_id = which(is_na)
mean_process03 = mean(CMP_subset$ManufacturingProcess03, na.rm = TRUE)
CMP_subset$ManufacturingProcess03[na_id] = mean_process03
```

Knowledge Check 1

Exercise 1

Module completion checklist

Objective	Complete
Load and evaluate the dataset	✔
Address missing values in data	✔
Manipulate data types and structures using flow control structures (`for` loops, conditionals,etc)	
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	

So, what is an algorithm?

- An algorithm is a process or set of rules to be followed in order to achieve a particular goal.
- In an algorithm for doing laundry, let's assume that you have the objects needed for the laundry: bleach, fabric softener, and any other items.
- The first step is you standing before a washing machine with your laundry and those items.
- The last state is a set of clean, neatly folded laundered clothes.

Control structures and functions

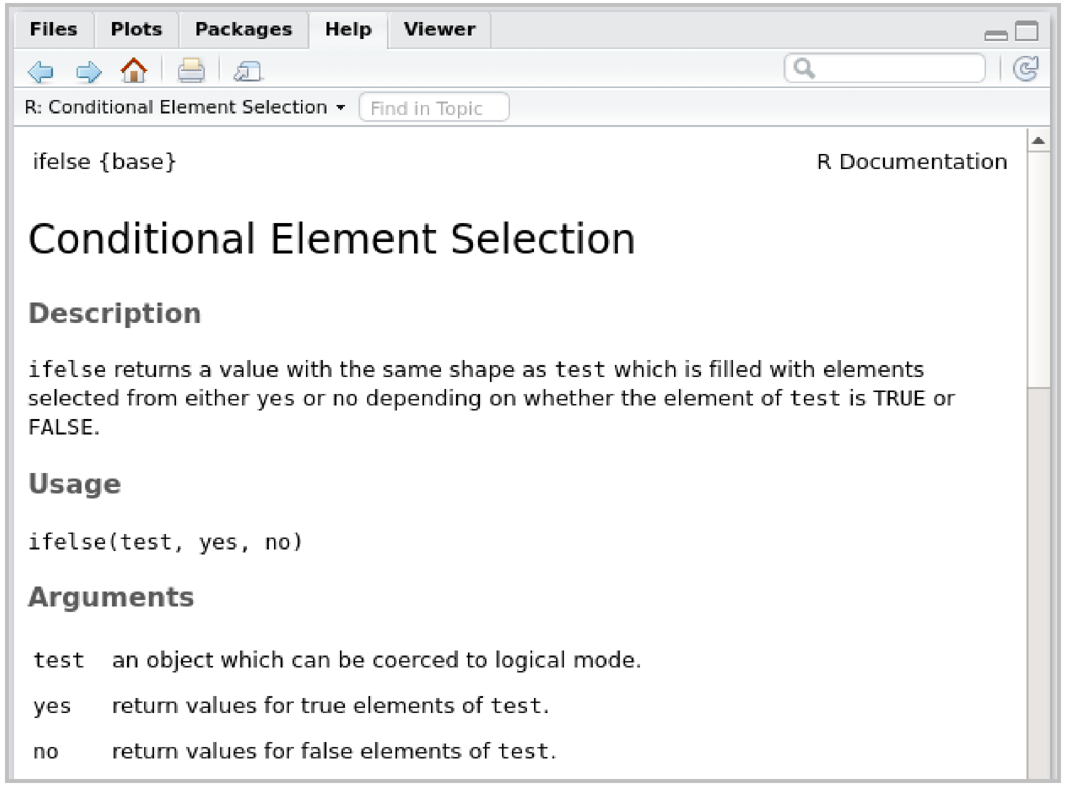
- No introduction to any programming language is complete without learning about control structures and functions
- If you understand the data types, basic data structures, control structures, and function definitions you will be able to complete most of the tasks related to problem solving using programming languages
- We will introduce you to
  - Writing conditional statements using `if`, `if...else`, and `ifelse`
  - Writing loops using `for`
  - Writing function definitions using `function`

Conditionals: `ifelse` function

- Conditionals are used to decide the flow of execution based on different conditions

```
?ifelse
```

- The simplest conditional is `ifelse` function. It has 3 arguments:
  - The **condition** for which we are testing (i.e. the `test`)
  - The **value** that is returned in case the condition specified is met
  - The **value** that is returned in case the condition specified is NOT met
- `ifelse` function must return a value



Ifelse example

- Let's say we want to take `Yield` from the `CMP` dataset and convert it to either `above average` or `below average`
- We can use `ifelse` here. Let's demonstrate:

```
meanCMP_yield = mean(CMP$Yield)

CMP$new_yield = ifelse(CMP$Yield >= meanCMP_yield,    #<- if CMP$Yield is greater than
                                                      #   or equal to the mean of Yield
                       "above_average",               #<- Then new_yield = above average
                       "below_average")               #<- Else new_yield = below average


head(CMP[,c("Yield","new_yield")])
```

```
  Yield     new_yield
1 38.00 below_average
2 42.44 above_average
3 42.03 above_average
4 41.42 above_average
5 42.49 above_average
6 43.57 above_average
```

Loops: `for` loop

- A `for` loop is used when we have a **finite** set of distinct repeated actions
- It has an explicit `start` and `end`
- Arguments for a `for` loop can take several forms, but the most common includes:
  - An arbitrary `counter` or `index` variable
  - The `in` word to indicate that the counter is an element of a sequence on its right hand side
  - A `sequence` of indices through which to **loop** defined in the `start:end` format

Loops: `for` loop

- Here is a basic example of a `for` loop

```
# Basic for loop.
for(i in 1:num_of_repetitions){
  perform action on element at index i }
```

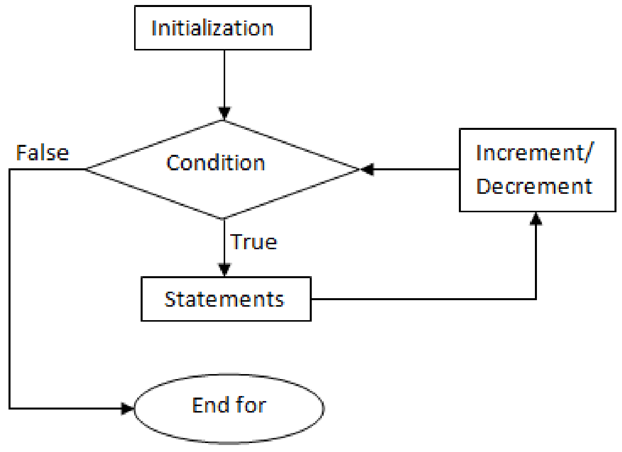


fig: Flowchart for for loop

- *Index `i` is an arbitrary letter/word, which we use to let the loop know which element is **current**. We pass that variable (i.e. index) to the data object so we isolate the work to be done ONLY on the **current** (i.e. `i-th`) element of the object.*

Loops: `for` loop

- We can identify the start and end points of the loop in several ways:
  - give the numbers in the index
  - give variables set equal to index numbers
  - go from 1 to length of list
- Let's make a 'for' loop to print only the variable names that start at index '3' and end at index '6'

```
CMP_subset_variables = colnames(CMP_subset)

# Adjust the start index.
seq_start = 3

# Adjust the end index.
seq_end = 6

# Loop through just a subset
# of the variable names.
for(i in seq_start:seq_end){
  print(CMP_subset_variables[i])
}
```

```
[1] "BiologicalMaterial02"
[1] "BiologicalMaterial03"
[1] "ManufacturingProcess01"
[1] "ManufacturingProcess02"
```

- In this case, we only wanted to print the variable names that start at index `3` and end at index `6`, so we only needed to adjust the `start` and `end` indices in the `for` loop

Functions in R

- **Functions** are chunks of code that allow you to:
  - Generalize your code so it can be re-used later
- They make your code:
  - More **abstract** so it can be used with different data and/or parameters
  - More **modular** so it can be used as a part of another larger chunk of code, script, or even a program
  - **Clean**, as they isolate actions performed and allow you to trace the flow of your code with ease

```
# Basic function with no arguments.
function(){
  perform action
}

# Basic function with 1 argument.
function(argument){
  perform action given argument
}

# Basic function with 2 (or more) arguments.
function(argument1, argument2){
  perform action given argument1, argument2
}
```

Functions in R: function without arguments

```
# Make a function that prints "Hello" and
# assign it to `PrintHello` variable.

PrintHello = function(){ #<- declare function
  print("Hello!")        #<- perform action
}

# Invoke function by calling `PrintHello()`.
PrintHello()
```

```
[1] "Hello!"
```

The most basic function is one that takes no arguments.

The function definition consists of 2 main components:

1. The chosen function name set equal to the `function` keyword followed by empty `()`
2. The body of the function that is defined within the `{}` can either:
   i. Perform an **action**
   ii. Return a specific **value**

Functions in R: function with arguments

```
# Make a function that prints "Hello, [name]".
PrintHello = function(name){     #<- Add `name` argument to function declaration

  # Save message to print to a variable.
  hello_name = paste0("Hello ", #<- concatenate "Hello "
                      name,      #<-  with the `name` from function argument, and
                      "!")       #<-  with the remainder of the message to print

  print(hello_name)              #<- print message
}

# Invoke function by calling `PrintHello([name])`.
PrintHello("User")
```

```
[1] "Hello User!"
```

Functions in R: function with arguments

```r
# Make function that rounds to the first `n` digits of `pi`.
GetPi = function(n){              #<- Add `n` argument to function declaration

  pi_num = round(3.14159265359, #<- Round `pi`
                 n)              #<-   to `n` digits
  return(pi_num)
}

# Invoke function by calling `GetPi([n])`.
GetPi(3)
```

```
[1] 3.142
```

Functions in R: wrapping it all into a function

- To define a function, we need to assign it to a variable (i.e. `ImputeNAsWithMean`) and add an argument to `()`
- We then need to substitute every instance of the specific dataset name with our argument (i.e. `dataset`)
- We need to `return` the updated `dataset` at the end of the function

Functions in R

```
ImputeNAsWithMean = function(dataset){

  for(i in 1:ncol(dataset)){
    is_na = is.na(dataset[, i])
    if(any(is_na)){
      na_ids = which(is_na)
      var_mean = mean(dataset[, i],
                      na.rm = TRUE)
      dataset[na_ids, i] = var_mean
      message = paste0(
        "NAs substituted with mean in
",
        colnames(dataset)[i])
      print(message)
    }
  }
  return(dataset)
}
```

Congratulations on creating your first function in R!

```
# Let's re-generate our subset again.
CMP_subset = CMP[, c(1:4, 14:16)]

# Let's test the function giving the `CMP_subset` as the
argument.
CMP_subset_imputed = ImputeNAsWithMean(CMP_subset)
```

```
[1] "NAs substituted with mean in ManufacturingProcess01"
[1] "NAs substituted with mean in ManufacturingProcess02"
[1] "NAs substituted with mean in ManufacturingProcess03"
```

```
# Inspect the structure.
str(CMP_subset_imputed)
```

```
'data.frame':   176 obs. of  7 variables:
 $ Yield                 : num  38 42.4 42 41.4 42.5 ...
 $ BiologicalMaterial01  : num  6.25 8.01 8.01 8.01 7.47
6.12 7.48 6.94 6.94 6.94 ...
 $ BiologicalMaterial02  : num  49.6 61 61 61 63.3 ...
 $ BiologicalMaterial03  : num  57 67.5 67.5 67.5 72.2 ...
 $ ManufacturingProcess01: num  11.2 0 0 0 10.7 ...
 $ ManufacturingProcess02: num  16.7 0 0 0 0 ...
 $ ManufacturingProcess03: num  1.54 1.54 1.54 1.54 1.54
...
```

Knowledge Check 2

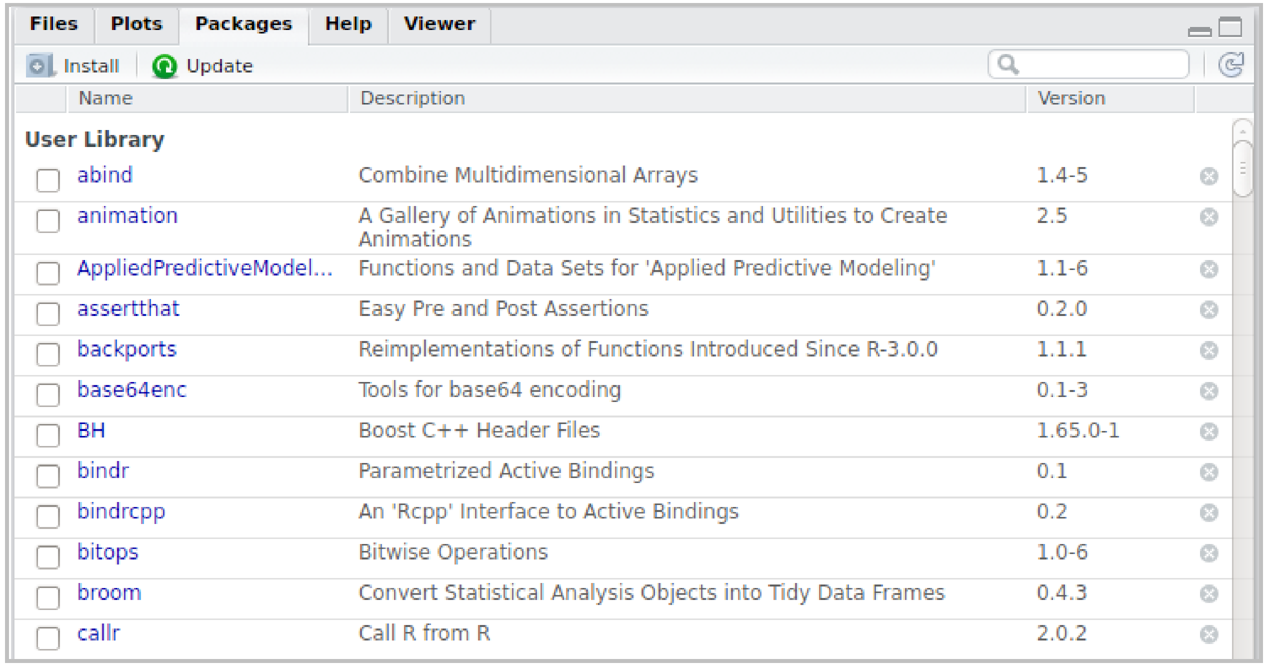
Exercise 2

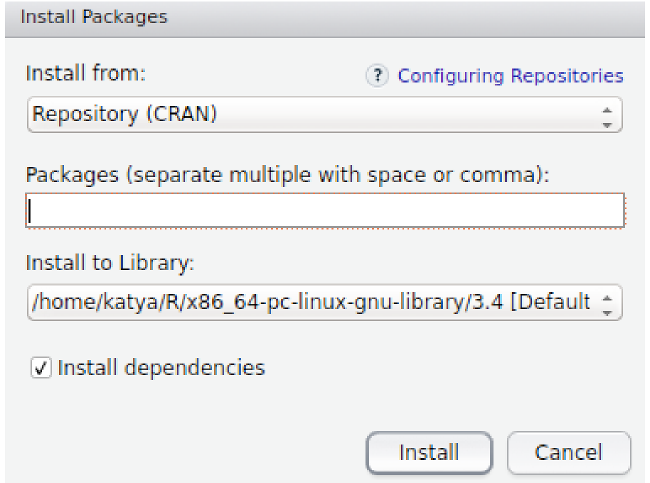
Module completion checklist

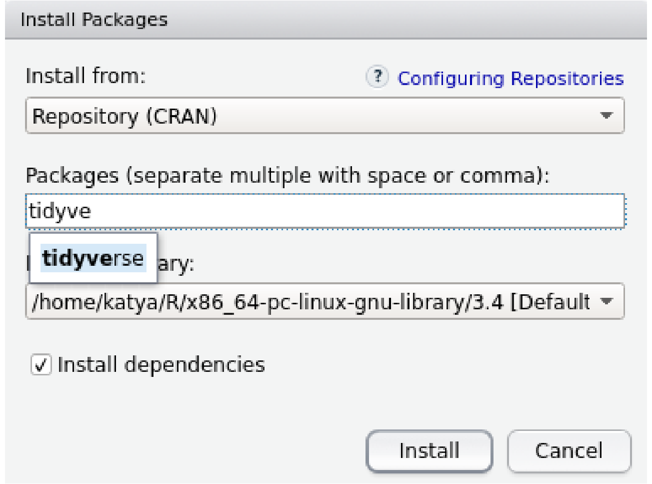
Objective	Complete
Load and evaluate the dataset	✔
Address missing values in data	✔
Manipulate data types and structures using flow control structures (`for` loops, conditionals,etc)	✔
Demonstrate installing a package and loading a library	
Define the six functions that provide verbs for the language of data manipulation, from the package dplyr	
Apply the filter function to subset data	
Rank data using the arrange function	

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

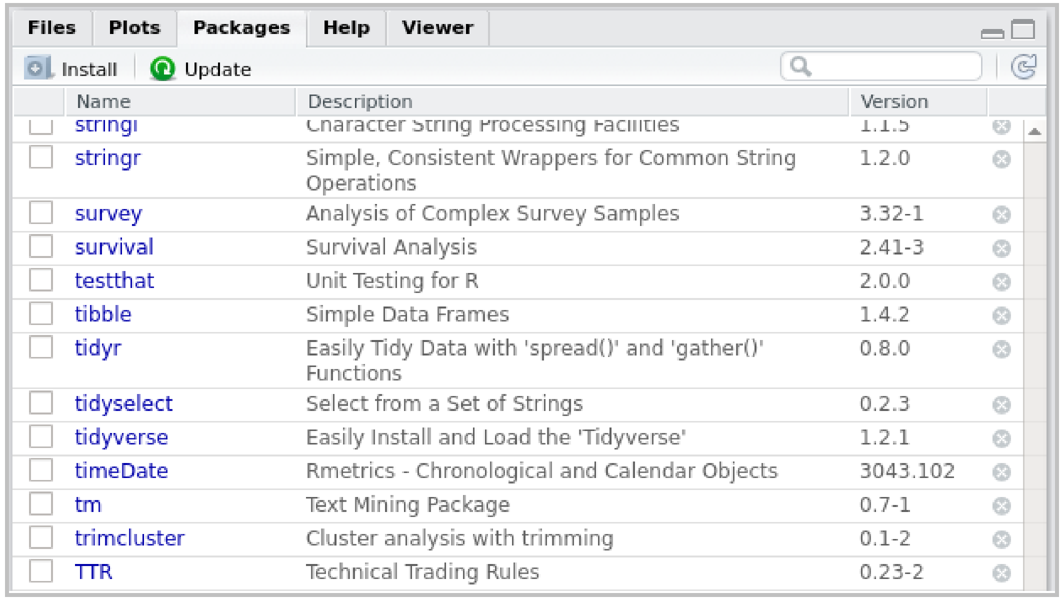




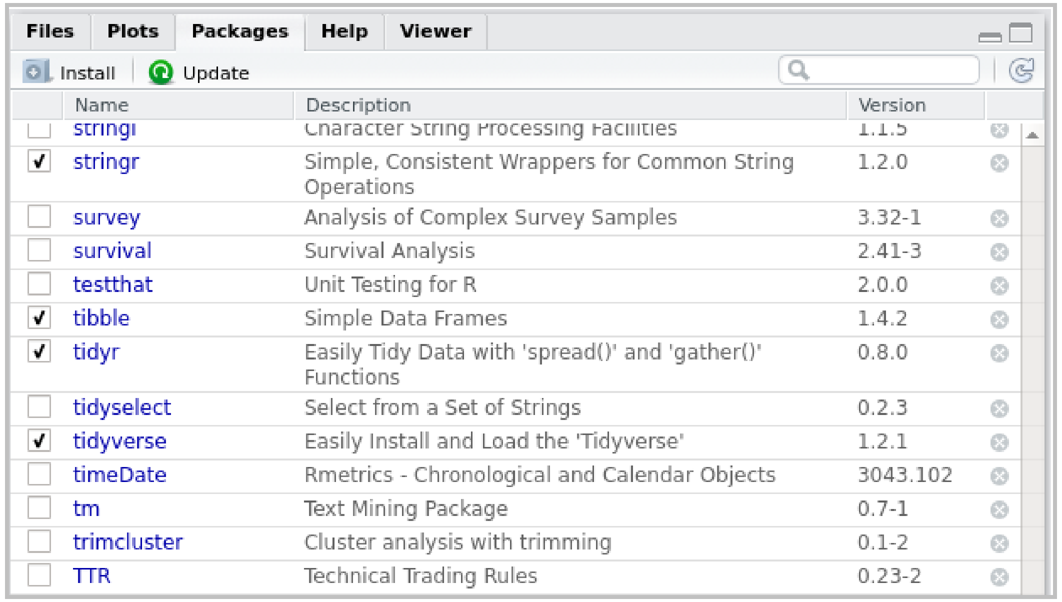


Installing packages: package explorer

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



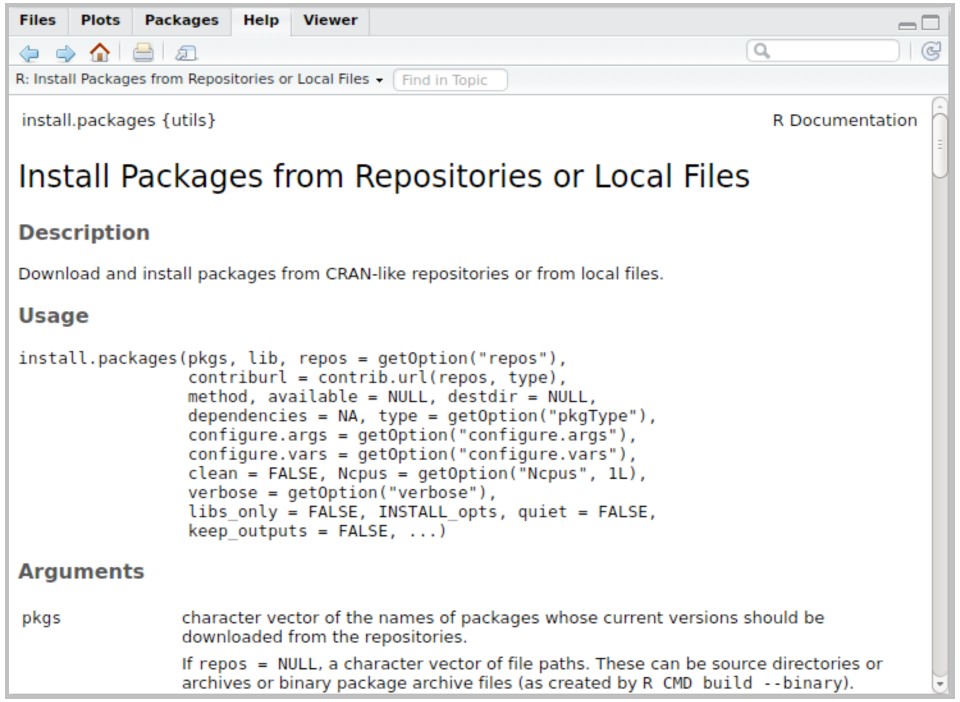
- To load the package into R's environment, check the box next to the name of your desired package



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, we can also use the function **`install.packages()`**
- For this function, we need to provide a single required argument: a character string corresponding to the `package name`

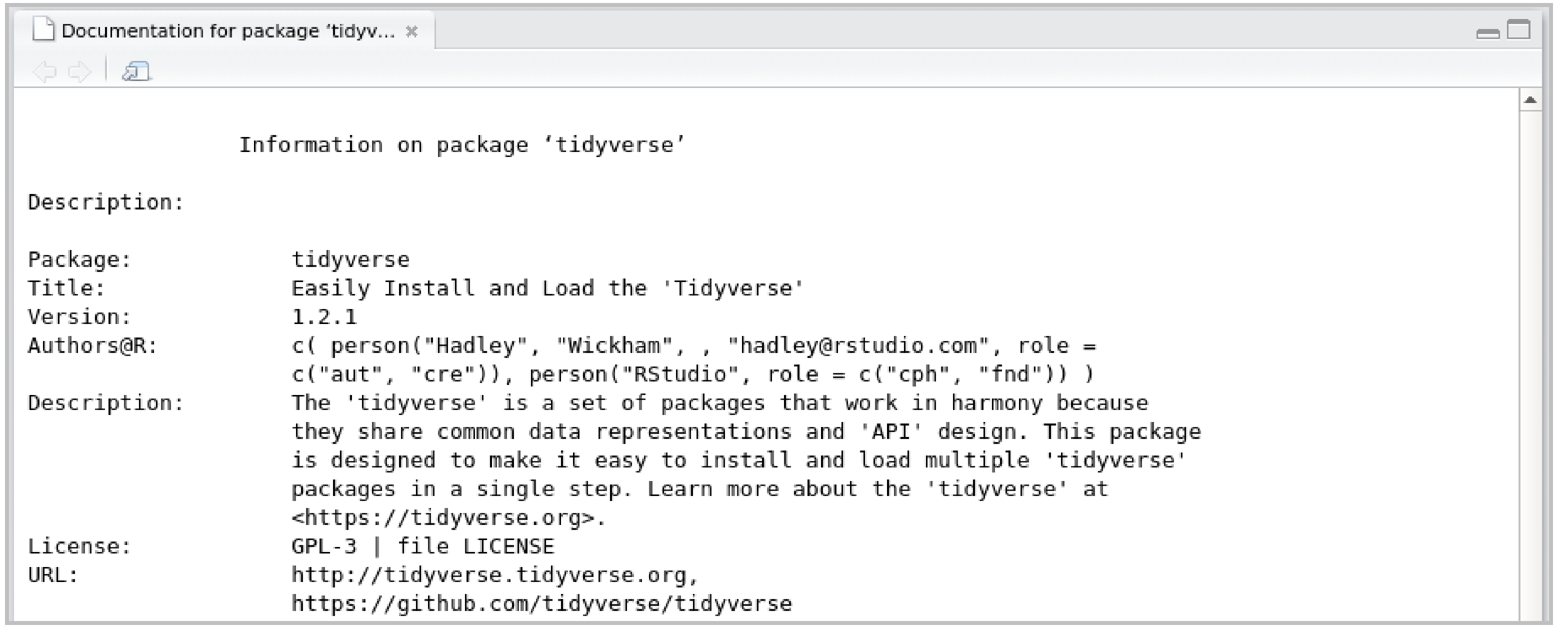
```
# Install package
?install.packages
```



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 a package with `help = "package name"`

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



```
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
```

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

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`

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`

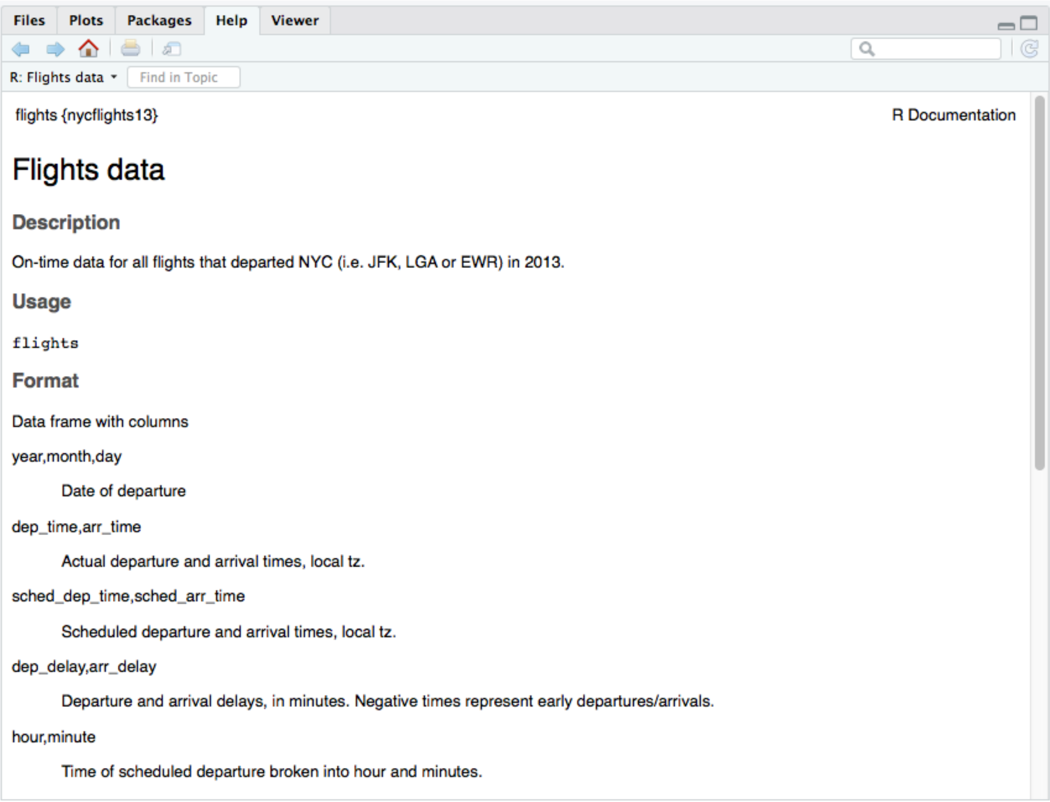
Data transformation

- 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
```



Basics of dplyr

- After getting familiar with our dataset, let's get back to the package we will be using, `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
`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
`summarize`	Collapse many values down to a single summary	All data types
`group_by`	Allows the above functions to operate on a dataset group by group	All data types

Framework of dplyr

- The framework of `dplyr` is as follows:
  - The first argument is a **dataframe**
  - The next arguments describe **what to do with the dataframe**, using the six key `dplyr` functions
  - The final result is a **new, transformed dataframe**
- We will now discuss how each of these six verbs work

Module completion checklist

Topic	Complete
Load and evaluate the dataset	✔
Address missing values in data	✔
Manipulate data types and structures using flow control structures (`for` loops, conditionals,etc)	✔
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	

Filter

- `filter` allows you to subset observations based on their values
- Basic use cases for the `filter` function include:
  - Subsetting the data to include flights from January 2013
  - Subsetting the data that contain missing values

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

# Use cases for `filter` function

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

filter {dplyr} R Documentation

Subset rows using column values

Description

The `filter()` function is used to subset a data frame, retaining all rows that satisfy your conditions. To be retained, the row must produce a value of `TRUE` for all conditions. Note that when a condition evaluates to `NA` the row will be dropped, unlike base subsetting with `[`.

Usage

```
filter(.data, ..., .preserve = FALSE)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.

- 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` dataframe 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 sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     1      517            515         2      830            819
 2  2013     1     1      533            529         4      850            830
 3  2013     1     1      542            540         2      923            850
 4  2013     1     1      544            545        -1     1004           1022
 5  2013     1     1      554            600        -6      812            837
 6  2013     1     1      554            558        -4      740            728
 7  2013     1     1      555            600        -5      913            854
 8  2013     1     1      557            600        -3      709            723
 9  2013     1     1      557            600        -3      838            846
10  2013     1     1      558            600        -2      753            745
# … with 26,994 more rows, and 11 more variables: 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>
```

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, year == 2013)

# View your output.
filter_flights
```

```
# A tibble: 27,004 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     1      517            515         2      830            819
 2  2013     1     1      533            529         4      850            830
 3  2013     1     1      542            540         2      923            850
 4  2013     1     1      544            545        -1     1004           1022
 5  2013     1     1      554            600        -6      812            837
 6  2013     1     1      554            558        -4      740            728
 7  2013     1     1      555            600        -5      913            854
 8  2013     1     1      557            600        -3      709            723
 9  2013     1     1      557            600        -3      838            846
10  2013     1     1      558            600        -2      753            745
# … with 26,994 more rows, and 11 more variables: 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>
```

Filter options

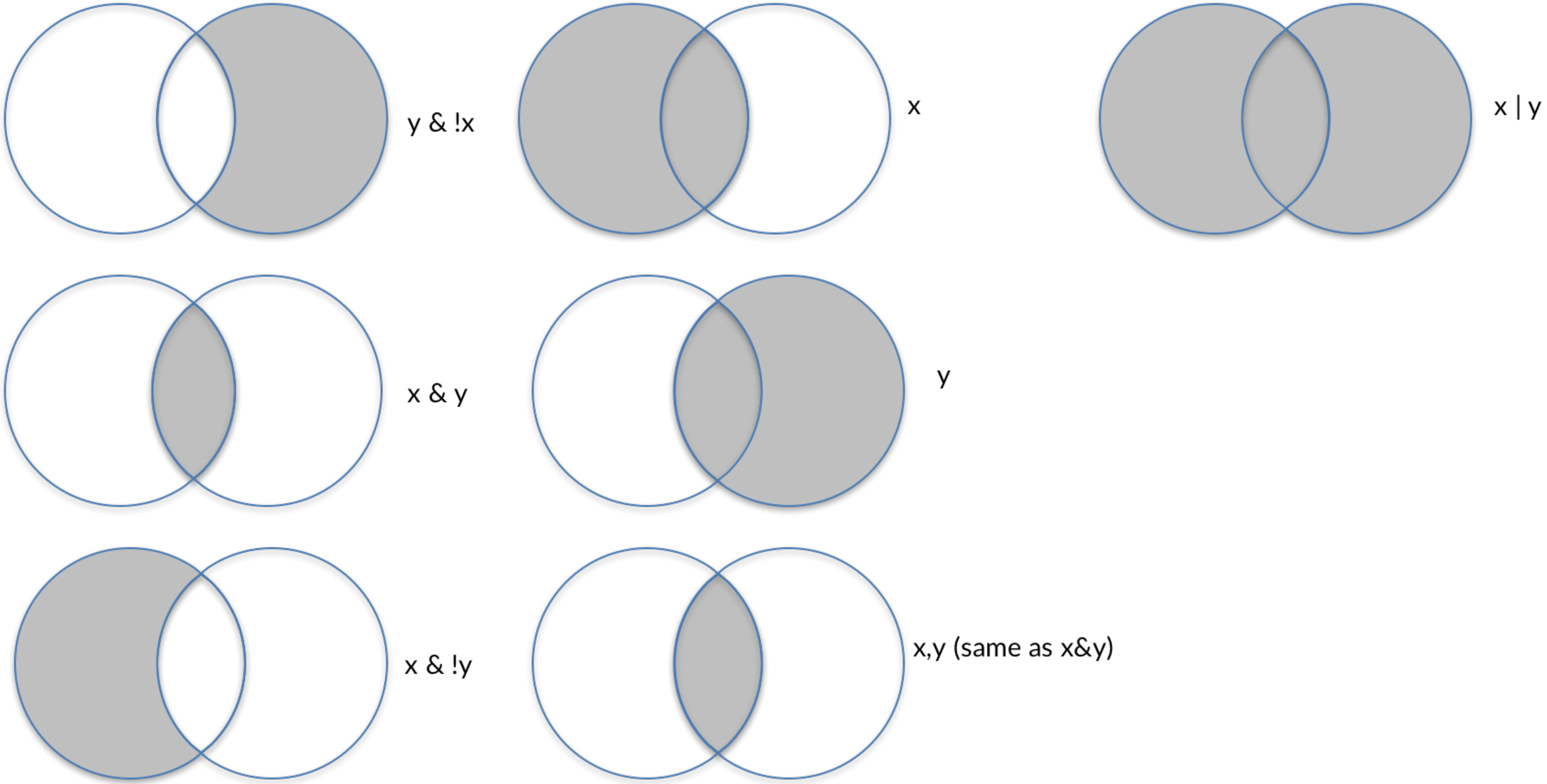
- You can use the standard filtering operations when working with integer data types:

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:

<table>
<tr><th>Operation</th><th>Use Case</th><th>Example</th></tr>
<tr><td>|</td><td>either can be true to satisfy</td><td>x == 4 | x == 12, x==2 | x==13</td></tr>
<tr><td>&</td><td>and, both need to be true</td><td>x == 4 & y == 2</td></tr>
<tr><td>!</td><td>not true, inverse selection</td><td>x != 4</td></tr>
<tr><td>%in%</td><td>value in the following list of values</td><td>x %in% c(4,16,32)</td></tr>
</table>

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 sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1    25       15           1815       360      208           1958
 2  2013     1    25       17           2249        88      119           2357
 3  2013     1    25       26           1850       336      225           2055
 4  2013     1    25      123           2000       323      229           2101
 5  2013     1    25      123           2029       294      215           2140
 6  2013     1    25      456            500        -4      632            648
 7  2013     1    25      519            525        -6      804            820
 8  2013     1    25      527            530        -3      820            829
 9  2013     1    25      535            540        -5      826            850
10  2013     1    25      539            540        -1     1006           1017
# … with 912 more rows, and 11 more variables: 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>
```

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

Filter - examples of logical operators (cont-d)

- 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 sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013    10     1      447            500       -13      614            648
 2  2013    10     1      522            517         5      735            757
 3  2013    10     1      536            545        -9      809            855
 4  2013    10     1      539            545        -6      801            827
 5  2013    10     1      539            545        -6      917            933
 6  2013    10     1      544            550        -6      912            932
 7  2013    10     1      549            600       -11      653            716
 8  2013    10     1      550            600       -10      648            700
 9  2013    10     1      550            600       -10      649            659
10  2013    10     1      551            600        -9      727            730
# … with 299,587 more rows, and 11 more variables: 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>
```

- Here we are looking for all flights that are **not in January** and **not on the 25th**; total number of rows should be **299,597**

Filter - examples of logical operators (cont-d)

```
# 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 sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1    25       15           1815       360      208           1958
 2  2013     1    25       17           2249        88      119           2357
 3  2013     1    25       26           1850       336      225           2055
 4  2013     1    25      123           2000       323      229           2101
 5  2013     1    25      123           2029       294      215           2140
 6  2013     1    25      456            500        -4      632            648
 7  2013     1    25      519            525        -6      804            820
 8  2013     1    25      527            530        -3      820            829
 9  2013     1    25      535            540        -5      826            850
10  2013     1    25      539            540        -1     1006           1017
# … with 1,873 more rows, and 11 more variables: arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

- This is a combination of `&` and `%in%` subsetting **all flights from January and February** that are **on the 25th**; number of rows should be **1,883**

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 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

# 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
```

Module completion checklist

Topic	Complete
Load and evaluate the dataset	✔
Address missing values in data	✔
Manipulate data types and structures using flow control structures (`for` loops, conditionals,etc)	✔
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	

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

arrange {dplyr} R Documentation

Arrange rows by column values

Description

`arrange()` order the rows of a data frame rows by the values of selected columns.

Unlike other dplyr verbs, `arrange()` largely ignores grouping; you need to explicit mention grouping variables (or use `by_group = TRUE`) in order to group by them, and functions of variables are evaluated once per data frame, not once per group.

Usage

```
arrange(.data, ..., .by_group = FALSE)

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

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<`data-masking`> Variables, or functions or variables. 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

- 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, #<- 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 sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     1      517            515         2      830            819
 2  2013     1     1      533            529         4      850            830
 3  2013     1     1      542            540         2      923            850
 4  2013     1     1      544            545        -1     1004           1022
 5  2013     1     1      554            600        -6      812            837
 6  2013     1     1      554            558        -4      740            728
 7  2013     1     1      555            600        -5      913            854
 8  2013     1     1      557            600        -3      709            723
 9  2013     1     1      557            600        -3      838            846
10  2013     1     1      558            600        -2      753            745
# … with 336,766 more rows, and 11 more variables: 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 sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013    12     1       13           2359        14      446            445
 2  2013    12     1       17           2359        18      443            437
 3  2013    12     1      453            500        -7      636            651
 4  2013    12     1      520            515         5      749            808
 5  2013    12     1      536            540        -4      845            850
 6  2013    12     1      540            550       -10     1005           1027
 7  2013    12     1      541            545        -4      734            755
 8  2013    12     1      546            545         1      826            835
 9  2013    12     1      549            600       -11      648            659
10  2013    12     1      550            600       -10      825            854
# … with 336,766 more rows, and 11 more variables: 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

```
# 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 3

Exercise 3

Module completion checklist

Topic	Complete
Load and evaluate the dataset	✔
Address missing values in data	✔
Manipulate data types and structures using flow control structures (`for` loops, conditionals,etc)	✔
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	✔

Summary

- Today, we loaded and inspected the CMP dataset, identifying NA values and imputing values for the missing data
- We also looked into various flow control structures and created our first **loop** function together in R
- We also learned how to install packages and used some of the functions from `tidyverse` package to perform data wrangling
- **In our next module**, you will learn more about data cleaning in R and some techniques to **visualize** our data and make our analysis results more accessible.
- **Stay excited!**

This completes our module

Congratulations!