SMU
SINGAPORE MANAGEMENT UNIVERSITY

Programming with Data

Session 3: R Programming (II)

Dr. Wang Jiwei
Master of Professional Accounting

Logical expressions

Why use logical expressions?

- We just saw an example in our subsetting function
  - `earnings < 20000`
- Logical expressions give us more control over the data
- They let us easily create logical vectors for subsetting data

```
df$earnings
```

```
## NULL
```

```
df$earnings < 20000
```

```
## logical(0)
```

Logical operators

`==` `!=` `>` `<` `>=` `<=` `!` `|` `&`

- Equals: `==`
  - `2 == 2` → TRUE
  - `2 == 3` → FALSE
  - `'dog'=='dog'` → TRUE
  - `'dog'=='cat'` → FALSE
- Not equals: `!=`
  - The opposite of `==`
  - `2 != 2` → FALSE
  - `2 != 3` → TRUE
  - `'dog'!='cat'` → TRUE
- Comparing strings is done character by character

Logical operators

SMU SINGAPORE MANAGEMENT UNIVERSITY School of Accountancy

`== != > < >= <= ! | &`

- Greater than: `>`
  - `2 > 1` → TRUE
  - `2 > 2` → FALSE
  - `2 > 3` → FALSE
  - `'dog'>'cat'` → TRUE
- Less than: `<`
  - `2 < 1` → FALSE
  - `2 < 2` → FALSE
  - `2 < 3` → TRUE
  - `'dog'<'cat'` → FALSE
- Greater than or equal to: `>=`
  - `2 >= 1` → TRUE
  - `2 >= 2` → TRUE
  - `2 >= 3` → FALSE
- Less than or equal to: `<=`
  - `2 <= 1` → FALSE
  - `2 <= 2` → TRUE
  - `2 <= 3` → TRUE

Logical operators

- Not: `!`
  - This simply inverts everything
  - `!TRUE` → FALSE
  - `!FALSE` → TRUE
- And: `&`
  - `TRUE & TRUE` → TRUE
  - `TRUE & FALSE` → FALSE
  - `FALSE & FALSE` → FALSE
- Or: `|` (pipe, same key as '\')
  - Note that `|` is evaluated after all `&`s
  - `TRUE | TRUE` → TRUE
  - `TRUE | FALSE` → TRUE
  - `FALSE | FALSE` → FALSE
- You can mix in parentheses for grouping as needed

Examples for logical operators

- How many tech firms had >$10B in revenue in 2017?

```
sum(tech_df$revenue > 10000)
```

```
## [1] 46
```

- How many tech firms had >$10B in revenue but had negative earnings in 2017?

```
sum(tech_df$revenue > 10000 & tech_df$earnings < 0)
```

```
## [1] 4
```

Examples for logical operators

- Who are those 4 with high revenue and negative earnings?

```
columns <- c("conm", "tic", "earnings", "revenue")
tech_df[tech_df$revenue > 10000 & tech_df$earnings < 0, columns]
```

```
##                                conm   tic  earnings  revenue
## 2100                   CORNING INC   GLW  -497.000 10116.00
## 2874  TELEFONAKTIEBOLAGET LM ERICS  ERIC -4307.493 24629.64
## 11804        DELL TECHNOLOGIES INC 7732B -3728.000 78660.00
## 23377                    NOKIA CORP   NOK -1796.087 27917.49
```

Other special values

- We know `TRUE` and `FALSE` already
  - Note that `FALSE` can be represented as 0
  - Note that `TRUE` can be represented as any non-zero number
- There are also:
  - `Inf`: Infinity, often caused by dividing something by 0
  - `NaN`: "Not a number," likely that the expression 0/0 occurred
  - `NA`: A missing value, usually *not* due to a mathematical error
  - `NULL`: Indicates a variable has nothing in it
- We can check for these with:
  - `is.inf()`
  - `is.nan()`
  - `is.na()`
  - `is.null()`

if ... else

- Conditional statements (used for programming)

```
# cond1, cond2, etc. can be any logical expression
if(cond1) {
  # Code runs if cond1 is TRUE
} else if (cond2) { # Can repeat 'else if' as needed
  # Code runs if this is the first condition that is TRUE
} else {
  # Code runs if none of the above conditions TRUE
}
```

Other uses

- Vectorized conditional statements using `ifelse()`
  - If else takes 3 vectors and returns 1 vector
    - A vector of TRUE or FALSE
    - A vector of elements to return from when TRUE
    - A vector of elements to return from when FALSE

```
# Outputs odd for odd numbers and even for even numbers
even <- rep("even", 5)
odd <- rep("odd", 5)
numbers <- 1:5
ifelse(numbers %% 2, odd, even)
```

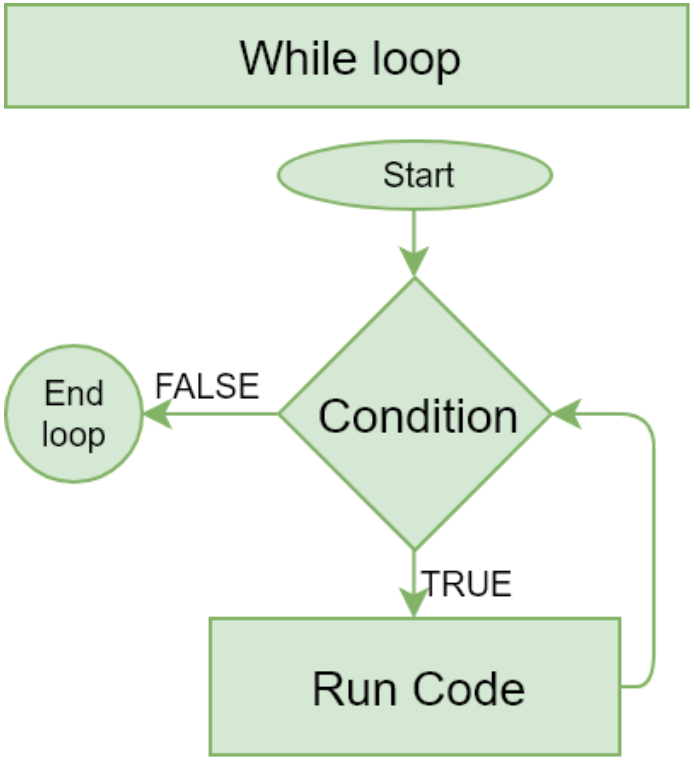
```
## [1] "odd"  "even" "odd"  "even" "odd"
```

Practice: Subsetting df

- This practice focuses on subsetting out potentially interesting parts of our data frame
  - We will also see which of Goldman, JPMorgan, and Citigroup, in which year, had the lowest earnings since 2010
- Do Exercise 5 on the following R practice file:
  - R Practice

Loops with control structure

Looping: While loop

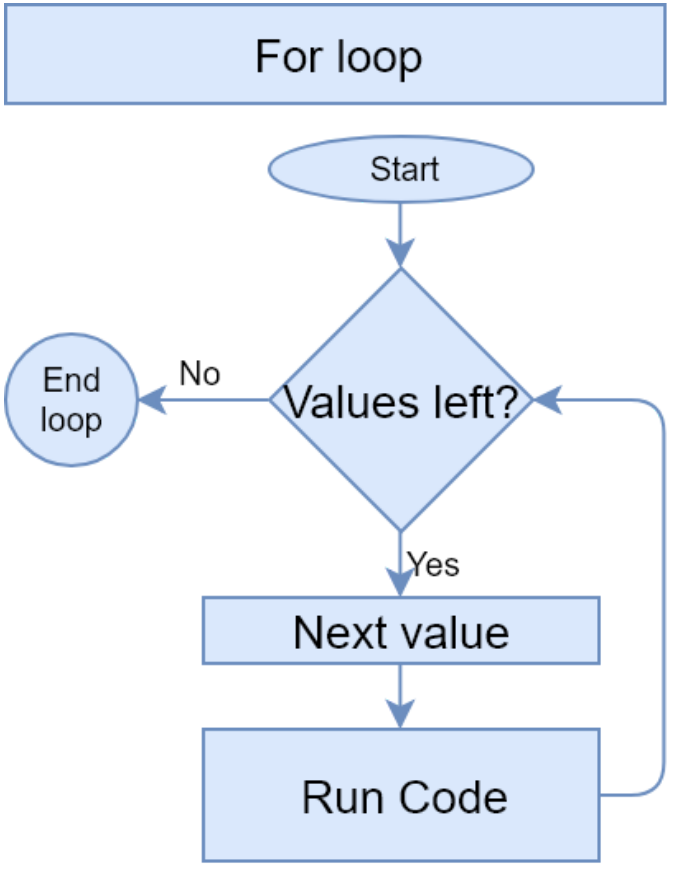


- A `while()` loop executes code repeatedly until a specified condition is `FALSE`
- An index shall be initiated before the `while` loop, and it must be changed within the loop, otherwise the loop will never end.

```
i <- 0
while(i < 5) {
  print(i)
  i <- i + 2
}
```

```
## [1] 0
## [1] 2
## [1] 4
```

Looping: For loop



- A `for()` loop executes code repeatedly until a specified condition is `FALSE`, while increamenting a given variable

```
for(i in c(0, 2, 4)) {
  print(i)
}
```

```
## [1] 0
## [1] 2
## [1] 4
```

Dangers of looping in R

- Loops in R are very slow -- one calculation at a time but R is best for many calculations at once via vectorization or matrix algebra
- `Sys.time()` to return the current system time

```
# Profit margin, all US tech firms
start <- Sys.time()
margin_1 <- rep(0,length(tech_df$ni))
for(i in seq_along(tech_df$ni)) {
  margin_1[i] <- tech_df$earnings[i] /
                 tech_df$revenue[i]
}
end <- Sys.time()
time_1 <- end - start
time_1
```

```
## Time difference of 0.00999999 secs
```

```
# Profit margin, all US tech firms
start <- Sys.time()
margin_2 <- tech_df$earnings /
            tech_df$revenue
end <- Sys.time()
time_2 <- end - start
time_2
```

```
## Time difference of 0.001999855 secs
```

Dangers of looping in R

- Loops in R are very slow -- one calculation at a time but R is best for many calculations at once via vectorization or matrix algebra

```
# Are these calculations identical?
identical(margin_1, margin_2)
```

```
## [1] TRUE
```

```
# How much slower is the loop?
paste(as.numeric(time_1) / as.numeric(time_2), "times")
```

```
## [1] "5.00035765379113 times"
```

Functions and packages

Help functions

- There are two equivalent ways to quickly access help files:
  - `?` and `help()`
  - Usage to get the help file for `data.frame()`:
    - `?data.frame`
    - `help(data.frame)`
- To see the options for a function, use `args()`

```
args(data.frame)
```

```
## function (..., row.names = NULL, check.rows = FALSE, check.names = TRUE,
##     fix.empty.names = TRUE, stringsAsFactors = FALSE)
## NULL
```

A note on using functions

```
args(data.frame)
```

```
## function (..., row.names = NULL, check.rows = FALSE, check.names = TRUE,
##     fix.empty.names = TRUE, stringsAsFactors = FALSE)
## NULL
```

- The `...` represents a series of inputs
  - In this case, inputs like `name=data`, where name is the column name and data is a vector
- The `_____ = _____` arguments are options for the function
  - The default is prespecified, but you can overwrite it
    - eg, you may change `stringsAsFactors` from FALSE (default) to TRUE
- Options can be very useful or save us a lot of time!
- You can always find them by:
  - Using the `?` command
  - Checking other documentation like www.rdocumentation.org
  - Using the `args()` function

Packages in R

- R packages are collections of functions and data sets developed by the community.
- Most R packages are stored on the offcial CRAN repository and can be installed within the RStudio directly
- Alternatively, you may download the package to local disk and use RStudio or command `install.packages(file.choose(), repos=NULL)` to install it

```
# To install the tidyverse package which will be used for this course
# tidyverse is a collection of useful packages in R
# https://www.tidyverse.org/
install.packages("tidyverse")

# or to install multiple packages in one go:
install.packages(c("ggplot2", "dplyr", "magrittr"))
```

- Load packages using `library()`
  - Need to do this each time you open a new instance of R

```
# Load the tidyverse package
library(tidyverse)
```

Pipe notation

SMU SINGAPORE MANAGEMENT UNIVERSITY School of Accountancy

> Pipe: output from the left as an input to the right directly.

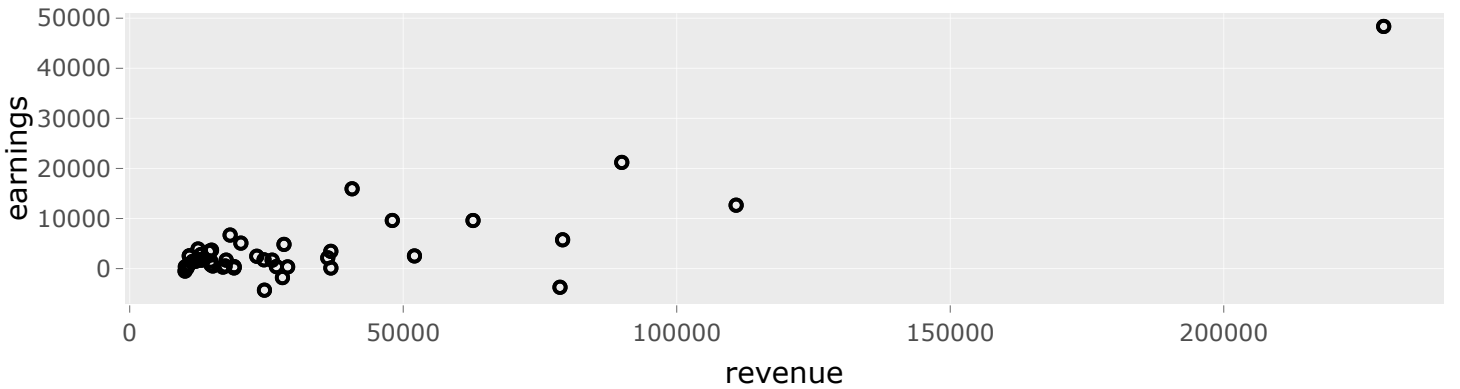
- The Base R (ie, without any external package) introduced the official pipe notation `|>` as of R version 4.1 in 2021.
  - The New R Pipe
- But a more popular pipe notation has already been provided by the `package:magrittr`
  - Part of `package:tidyverse`, an extremely popular collection of packages
- Pipe notation is done using `%>%`
  - `Left %>% Right(arg2, ...)` is the same as `Right(Left, arg2, ...)`

> Piping can drastically improve code readability

Piping example

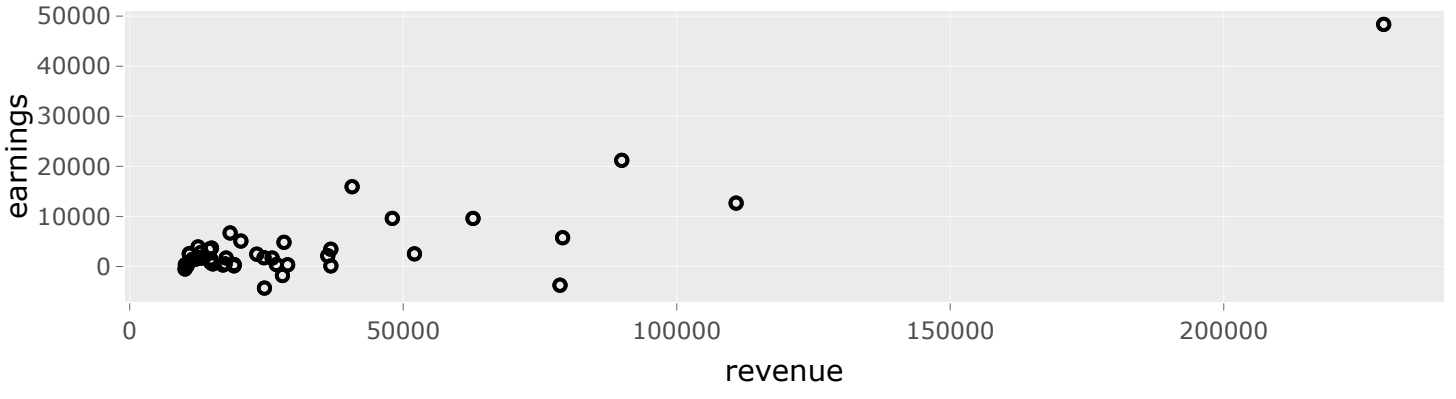
- Plot tech firms' earnings vs. revenue, >$10B in revenue

```
# %>% comes from magrittr and ggplot() comes from ggplot2, both part of tidyverse
# alternatively you may launch these two packages separately
# note that ggplot uses a special pipe notation "+"
library(tidyverse)
library(plotly)

plot <- tech_df %>%
  subset(revenue > 10000) %>%
  ggplot(aes(x = revenue, y = earnings)) +  # Adds point, and ticker
  geom_point(shape = 1, aes(text = sprintf("Ticker: %s", tic)))
ggplotly(plot)  # Makes the plot interactive
```



Without piping

```
library(tidyverse)
library(plotly)

plot <- ggplot(subset(tech_df, revenue > 10000),
               aes(x = revenue, y = earnings)) +
  geom_point(shape = 1, aes(text = sprintf("Ticker: %s", tic)))
ggplotly(plot)  # Makes the plot interactive
```



Practice: library usage

- This practice focuses on using an external library
  - We will also see which of Goldman, JPMorgan, and Citigroup, in which year, had the lowest earnings since 2010
- Do Exercise 6 on the following R practice file:
  - R Practice

> Note: The ~ indicates a formula the left side is the y-axis and the right side is the x-axis
>
> Note: The | tells lattice to make panels based on the variable(s) to the right

Math functions

- `sum()`: Sum of a vector
- `abs()`: Absolute value
- `sign()`: The sign of a number

```
vector = c(-2, -1, 0, 1, 2)
sum(vector)
```

```
## [1] 0
```

```
abs(vector)
```

```
## [1] 2 1 0 1 2
```

```
sign(vector)
```

```
## [1] -1 -1  0  1  1
```

Stats functions

- `mean()`: Calculates the mean of a vector
- `median()`: Calculates the median of a vector
- `sd()`: Calculates the sample standard deviation of a vector
- `quantile()`: Provides the *quartiles* of a vector
- `range()`: Gives the minimum and maximum of a vector
  - Related: `min()` and `max()`

```
quantile(tech_df$earnings)
```

```
##         0%        25%        50%        75%       100%
## -4307.4930   -15.9765     1.8370    91.3550 48351.0000
```

```
range(tech_df$earnings)
```

```
## [1] -4307.493 48351.000
```

Make your own functions!

- Use the `function()` function!
  - `my_func <- function(agruments) {code}`
  - recommended to explicitly use `return()` to specify what to return from the function.

> Simple function: Add 2 to a number

```
add_two <- function(n) {
  n + 2
}
add_two(500)
```

```
## [1] 502
```

```
add_two <- function(n) {
  return(n + 2)
}
add_two(500)
```

```
## [1] 502
```

Slightly more complex

```
mult_together <- function(n1, n2=0, square=FALSE) {
  if (!square) {
    return(n1 * n2)
  } else {
    return(n1 * n1)
  }
}

mult_together(5, 6)
```

```
## [1] 30
```

```
mult_together(5, 6, square = TRUE)
```

```
## [1] 25
```

```
mult_together(5, square = TRUE)
```

```
## [1] 25
```

Practice: Functions

- This practice focuses on making a custom function
  - Currency conversion between USD and SGD!
- Do Exercise 7 on the following R practice file:
  - R Practice

Challenging Practice

Define a function called `digits(n)` which returns the number of digits of a given integer number. For simplicity, we assume `n` is zero or positive integer, ie, n >= 0.

- if you call `digits(251)`, it should return `3`
- if you call `digits(5)`, it should return `1`
- if you call `digits(0)`, it should return `1`

For practice, you are required to use `if` conditions and `while` loops when necessary. You should use integer division `%/%` in the `while` loop to count the number of digits. You are not allowed to use functions such as `nchar()` and `floor()`.

Loops with `lapply()` functions

Loops with `lapply()`

You don't have to always write loops using `for` or `while`. There are a group of `lapply()` functions which can implement loops.

- `lapply()`: Loop over a list, evaluate a function on each element, and return a list
- there are some others too: `sapply()`; `mapply()`; `apply()`; `vapply()`; `tapply()`

Let's see the structure of `lapply()`. It extracts the function using `match.fun()`, checks whether it is a list (if not, convert to a list using `as.list()`) and finally loop internally in C code (`.Internal(lapply(X, FUN))`).

```
lapply
```

```
## function (X, FUN, ...)
## {
##     FUN <- match.fun(FUN)
##     if (!is.vector(X) || is.object(X))
##         X <- as.list(X)
##     .Internal(lapply(X, FUN))
## }
## <bytecode: 0x000000001ab7c2e8>
## <environment: namespace:base>
```

Apply a function over a list

`rnorm()` to generate normal distributed numbers (in a vector format) with default 0 mean and 1 standard deviations.

```
set.seed(1) # make random number generation reproducible
x_list <- list(a = rnorm(10000), b = rnorm(20000, 1, 5))
str(x_list)
```

```
## List of 2
##  $ a: num [1:10000] -0.626 0.184 -0.836 1.595 0.33 ...
##  $ b: num [1:20000] -3.02 -4.28 -4.18 -4.93 -1.5 ...
```

```
x_list_mean <- lapply(x_list, mean)
str(x_list_mean)
```

```
## List of 2
##  $ a: num -0.00654
##  $ b: num 1.01
```

```
x_list_mean_vector <- sapply(x_list, mean)
str(x_list_mean_vector)
```

```
##  Named num [1:2] -0.00654 1.00841
##  - attr(*, "names")= chr [1:2] "a" "b"
```

Apply a function over an array

`array()` are data objects which can store data in more than two dimensions which allows different data types. Recall that `matrix` is two-dimensional data with same data type and `dataframe` is two-dimensional data which allows different data types. `apply()` can evaluate a function over an array.

```
set.seed(1) # make random number generation reproducible
# create a 2-dimensional array (a matrix for this case)
x_array <- array(c(rnorm(10000), rnorm(20000, 1, 5)), dim = c(2, 10000))
str(x_array)
```

```
##  num [1:2, 1:10000] -0.626 0.184 -0.836 1.595 0.33 ...
```

```
# apply mean() on the first dimension, ie, rows of a matrix/dataframe
x_array_mean <- apply(x_array, 1, mean)
str(x_array_mean)
```

```
##  num [1:2] 0.467 0.506
```

```
# apply mean() on the second dimension, ie, columns of a matrix/dataframe
x_array_mean <- apply(x_array, 2, mean)
str(x_array_mean)
```

```
##  num [1:10000] -0.221 0.38 -0.245 0.613 0.135 ...
```

Managing dataframes with `dplyr`

Read files to data frames

The most popular file format among data analysts is the comma-separated values (CSV) file that uses a comma (`,`) to separate values. Each line of the file is a data record. Each record consists of one or more fields, separated by commas.

- you can save Excel file into CSV file

The simplest way to import smaller CSV is to use the `read.csv()` from the base R (ie, without any additional packages). Other functions include: `read.table()` (for .txt or a tab-delimited text file); `read.delim()`(for file with a separator that is different from a tab, a comma or a semicolon)

```
df <- read.csv("data/session_2.csv")
```

Other packages also have import files functions:

- readr::read_csv()
- data.table::fread()
- readxl::read_excel()
- other packages for other data formats such as JSON, HTML, SAS, STATA, etc

Single table functions

`package:dplyr` is part of the `package:tidyverse` which provides useful functions for data manipulation. A competing package is `package:data.table` which is more efficient for large dataset (I suggest > 1G)

- Rows:
  - `filter()` chooses rows based on column values.
  - `slice()` chooses rows based on location.
  - `arrange()` changes the order of the rows.
- Columns:
  - `select()` changes whether or not a column is included.
  - `rename()` changes the name of columns.
  - `mutate()` changes the values of columns and creates new columns.
  - `relocate()` changes the order of the columns.
- Groups of rows:
  - `summarize()` collapses a group into a single row.

Filter rows with `filter()`

`filter()` allows you to select a subset of rows in a data frame. The first argument is the dataframe. The second and subsequent arguments refer to variables within that dataframe, selecting rows where the expression is `TRUE`.

Select all rows with ticker = AAPL (Apple Inc.) and after 2013 fiscal year:

```
library(tidyverse)
df %>% filter(tic == "AAPL" & fyear > 2013)
```

```
##   gvkey datadate fyear indfmt consol popsrc datafmt  tic      conm curcd    ni
## 1  1690 20140930  2014   INDL      C      D     STD AAPL APPLE INC   USD 39510
## 2  1690 20150930  2015   INDL      C      D     STD AAPL APPLE INC   USD 53394
## 3  1690 20160930  2016   INDL      C      D     STD AAPL APPLE INC   USD 45687
## 4  1690 20170930  2017   INDL      C      D     STD AAPL APPLE INC   USD 48351
##     revt    cik costat   gind gsector  gsubind
## 1 182795 320193      A 452020      45 45202030
## 2 233715 320193      A 452020      45 45202030
## 3 215091 320193      A 452020      45 45202030
## 4 229234 320193      A 452020      45 45202030
```

This is roughly equivalent to this base R code:

```
df[df$tic == "AAPL" & df$fyear > 2013, ]
```

Choose rows with `slice()`

`slice()` is to select, remove, and duplicate rows by their (integer) locations.

```
df %>% slice(5:7)
```

```
##   gvkey datadate fyear indfmt consol popsrc datafmt tic     conm curcd   ni
## 1  1004 20150531  2014   INDL      C      D     STD AIR AAR CORP   USD 10.2
## 2  1004 20160531  2015   INDL      C      D     STD AIR AAR CORP   USD 47.7
## 3  1004 20170531  2016   INDL      C      D     STD AIR AAR CORP   USD 56.5
##     revt  cik costat   gind gsector  gsubind
## 1 1594.3 1750      A 201010      20 20101010
## 2 1662.6 1750      A 201010      20 20101010
## 3 1767.6 1750      A 201010      20 20101010
```

It is accompanied by a number of helpers for common use cases:

- `slice_head()` and `slice_tail()` select the first or last rows.
- `slice_sample()` randomly selects rows.
- `slice_min()` and `slice_max()` select rows with highest or lowest values of a variable.

Arrange rows with `arrange()`

`arrange()` is to reorder the rows by a set of column names:

```
df %>% arrange(conm, desc(fyear)) %>%
  head()
```

```
##    gvkey datadate fyear indfmt consol popsrc datafmt  tic             conm
## 1 122519 20170630  2017   INDL      C      D     STD FLWS 1-800-FLOWERS.COM
## 2 122519 20160630  2016   INDL      C      D     STD FLWS 1-800-FLOWERS.COM
## 3 122519 20150630  2015   INDL      C      D     STD FLWS 1-800-FLOWERS.COM
## 4 122519 20140630  2014   INDL      C      D     STD FLWS 1-800-FLOWERS.COM
## 5 122519 20130630  2013   INDL      C      D     STD FLWS 1-800-FLOWERS.COM
## 6 122519 20120630  2012   INDL      C      D     STD FLWS 1-800-FLOWERS.COM
##   curcd     ni     revt     cik costat   gind gsector  gsubind
## 1   USD 44.041 1193.625 1084869      A 255020      25 25502020
## 2   USD 36.875 1173.024 1084869      A 255020      25 25502020
## 3   USD 20.287 1121.506 1084869      A 255020      25 25502020
## 4   USD 15.372  756.345 1084869      A 255020      25 25502020
## 5   USD 12.321  735.497 1084869      A 255020      25 25502020
## 6   USD 17.646  716.257 1084869      A 255020      25 25502020
```

Select columns with `select()`

`select()` allows you to subset a data frame by column names (variables/features/predictors)

```
# Select columns by name
df %>% select(gvkey, tic, conm, fyear) %>%
  slice(1:3)
```

```
##   gvkey tic     conm fyear
## 1  1004 AIR AAR CORP  2010
## 2  1004 AIR AAR CORP  2011
## 3  1004 AIR AAR CORP  2012
```

```
# Select all columns between gvkey and conm (inclusive)
df %>% select(gvkey:conm)
# Select all columns except those from gvkey to conm (inclusive)
df %>% select(!(gvkey:conm))
# Select all columns ending with "d"
df %>% select(ends_with("d"))
```

Rename columns with `rename()`

`rename()` allows you to rename column names

```
# rename columns
df %>% select(gvkey, tic, conm, fyear) %>%
  rename(comp_name = conm) %>%  slice(1:3)
```

```
##   gvkey tic comp_name fyear
## 1  1004 AIR  AAR CORP  2010
## 2  1004 AIR  AAR CORP  2011
## 3  1004 AIR  AAR CORP  2012
```

Add new columns with `mutate()`

`mutate()` is to add new columns. `package:DT` helps to present larger dataset using the `datatable()` function.

```
library(DT)
df %>% mutate(margin = ni / revt) %>% slice(1:20) %>%
  select(gvkey, conm, tic, fyear, ni, revt, margin) %>%
  datatable(options = list(pageLength = 2), rownames = FALSE)
```

Show 2 entries Search:

gvkey	conm	tic	fyear	ni	revt	margin
1004	AAR CORP	AIR	2010	69.826	1775.782	0.0393212680385318
1004	AAR CORP	AIR	2011	67.723	2074.498	0.0326454882096777

Showing 1 to 2 of 20 entries

Previous 1 2 3 4 5 … 10 Next

Change column order with `relocate()`

`relocate()` uses a similar syntax as `select()` to move blocks of columns at once

```
df %>% relocate(tic:revt, .after = fyear) %>%
  tail()
```

```
##        gvkey datadate fyear  tic               conm curcd      ni    revt indfmt
## 72720 324684 20171231  2017 ASLN ASLAN PHARMACEUTIC   USD -39.892     0.0   INDL
## 72721 326688 20131231  2013  NVT NVENT ELECTRIC PLC   USD      NA      NA   INDL
## 72722 326688 20141231  2014  NVT NVENT ELECTRIC PLC   USD      NA      NA   INDL
## 72723 326688 20151231  2015  NVT NVENT ELECTRIC PLC   USD      NA      NA   INDL
## 72724 326688 20161231  2016  NVT NVENT ELECTRIC PLC   USD 259.100 2116.0   INDL
## 72725 326688 20171231  2017  NVT NVENT ELECTRIC PLC   USD 361.700 2097.9   INDL
##       consol popsrc datafmt     cik costat   gind gsector  gsubind
## 72720      C      D     STD 1722926      A 352010      35 35201010
## 72721      C      D     STD 1720635      A 201040      20 20104010
## 72722      C      D     STD 1720635      A 201040      20 20104010
## 72723      C      D     STD 1720635      A 201040      20 20104010
## 72724      C      D     STD 1720635      A 201040      20 20104010
## 72725      C      D     STD 1720635      A 201040      20 20104010
```

Summarise values with `summarise()`

`summarize()` collapses a data frame to a single row.

```
df %>% summarise(ni_mean = mean(ni, na.rm = TRUE))
```

```
##    ni_mean
## 1 263.1611
```

It's not that useful until we learn the `group_by()` verb in a future topic.

Summary of Session 3

For next week

- continue with your Datacamp and textbook (R Cookbook or R for Data Science)
- review today's code and pre-read next week's seminar notes
- complete the **Assignment 1** and submit on eLearn

R Coding Style Guide

Style is subjective and arbitrary but it is important to follow a generally accepted style if you want to share code with others. I suggest the The tidyverse style guide which is also adopted by Google with some modification

- Highlights of **the tidyverse style guide**:
  - *File names*: end with .R
  - *Identifiers*: variable_name, function_name, try not to use "." as it is reserved by Base R's S3 objects
  - *Line length*: 80 characters
  - *Indentation*: two spaces, no tabs (RStudio by default converts tabs to spaces and you may change under global options)
  - *Spacing*: x = 0, not x=0, no space before a comma, but always place one after a comma
  - *Curly braces {}*: first on same line, last on own line
  - *Assignment*: use `<-`, not `=` nor `->`
  - *Semicolon(;)*: don't use, I used once for the interest of space
  - *return()*: Use explicit returns in functions: default function return is the last evaluated expression
  - *File paths*: use relative file path "../../filename.csv" rather than absolute path "C:/mydata/filename.csv". Backslash needs `\\`

R packages used in this slide

This slide was prepared on 2021-08-29 from Session_3s.Rmd with R version 4.1.1 (2021-08-10) Kick Things on Windows 10 x64 build 18362 .

The attached packages used in this slide are:

```
##         DT     plotly    forcats    stringr      dplyr      purrr      readr
##     "0.18"  "4.9.4.1"    "0.5.1"    "1.4.0"    "1.0.7"    "0.3.4"    "2.0.1"
##      tidyr     tibble    ggplot2  tidyverse kableExtra      knitr
##    "1.1.3"    "3.1.3"    "3.3.5"    "1.3.1"    "1.3.4"     "1.33"
```