An Introduction to R and Rstudio

Dr. Irene Vrbik

University of British Columbia Okanagan

irene.vrbik@ubc.ca

What is R?

- R is a free and open source programming language for statistical computing and graphics.
- It is one of the most widely used programming languages for statistical analysis.
- R is popular in both in academia and industry:

Google	For advertising effectiveness and economic forecasting.
Twitter	For data visualization and semantic clustering
Uber	For statistical analysis

see here for a full list of companies using R.

Why learn R?

- One major attraction of `R` is its rich variety of libraries.
- New libraries/packages (akin to Python modules) are continuously being added to the official open-source repository called CRAN (Comprehensive R Archive Network).
- There currently over 12000 packages which you can see by name here.
- In addition, mathematical calculations (eg vector operations, matrix multiplication) can be done in base R (i.e. without having to install any pacakges).

R vs. Python

- Both Python and R are amongst the most popular languages for data analysis.
- While Python may be touted as a more general-purpose languages, `R` has been targeted towards for statisticians and data scientist.
- `R` therefore does much of the heavy lifting in terms of analysis and graphing.
- Here are some interesting articles comparing Python to `R`: guru99, datacamp, towardsdatascience.

Example

Filtering a dataset to be within a lower and upper bound and then calculating summary statistics.

Python

```python
for v in data:
    if lower <= v <= upper:
        maxdata = max(v, maxdata)
        mindata = min(v, mindata)
    sumdata += v
    count += 1
```

Example

Filtering a dataset to be within a lower and upper bound and then calculating summary statistics.

R
```r
new_data <- subset(data, x <= upper & x >= lower)
sumdata <- sum(new_data)
count <- length(new_data)
maxdata <- max(new_data)
mindata <- min(new_data)
```

Statistics Review: Types of Data

Before diving into the R specifics, a brief review of some topics in statistics are in order. Recall that we can generalize data into two types:

1. Qualitative (Categorical)
   1.1 Descriptions or groups
   1.2 Can be characters or numbers
   1.3 Observed and not measured
   1.4 i.e. names, labels, categories, properties
2. Quantitative (Numeric)
   2.1 Strictly numeric
   2.2 Can be measured
   2.3 i.e. height, weight, speed, counts, temperature, volume

Summaries of Data

- A numerical summary provides an overview of data to help understand it without examining all data values.
- A measure of centre and a measure of spread to describe quantitative data.
- There are no numerical summaries for qualitative data, only graphical summaries.

Measures of Centre

Definition (Mean)

The mean is the average of data values (sum of values divided by count):

$$\overline{y} = \frac{\sum_{i=1}^{n} y_i}{n}$$

Definition (Median)

The median is the value at which half of the data lies above that value and half lies below it.

1. Odd number of observations: $\tilde{y}$ is the kth value where $k = \frac{n+1}{2}$.
2. Even number of observations: $\tilde{y}$ is the mean of the kth and $(k+1)$ terms, where $k = \frac{n}{2}$.

Example

Let the data be given by $y = \{1, 3, 3, 7, 9\}$. The mean is

$$\bar{y} = \frac{1 + 3 + 3 + 7 + 9}{5} = 4.6$$

and the median is

$$\tilde{y} = 3$$

for which R provides `mean()` and `median()`:

```
> mean(y)
[1] 4.6
> median(y)
[1] 3
```

Measures of Spread

A measure of spread indicates how far apart the values are.

Definition (Variance)

Variance is the sum of the squares of each data point's distance from the mean:

$$s^2 = \frac{\sum_{i=1}^{n}(y_i - \overline{y})^2}{n-1} = \frac{\sum_{i=1}^{n} y_i^2 - n\overline{y}^2}{n-1}$$

Definition (Standard Deviation)

Standard Deviation is the square root of the variance: $s = \sqrt{s^2}$

Definition (Range)

Range is the maximum value minus minimum value: $\max(y) - \min(y)$.

Example (Standard Deviation)

Let $y = (1, 3, , 3, 7, 9)$ then the variance is

$$s^2 = \frac{(1 + 9 + 9 + 49 + 81) - (5 \cdot 4.6^2)}{5 - 1} = 10.8$$

and therefore the standard deviation is

$$s = 3.286$$

for which R provides `mean()` and `median()`:

```
> var(y)
[1] 10.8
> median(y)
[1] 3.286335
```

Question

Let $y = (1, 3, 4, 7, 11, 28)$. How many of the are following statements are true?

1. $\overline{y} = \tilde{y}$
2. $\overline{y} = 9$
3. $s^2 = 3.5$
4. $\mathrm{range} = 6$.

A) 0 B) 1 C) 2 D) 3 E) 4

Question

Let $y = (1, 3, 4, 7, 11, 28)$. How many of the are following statements are true?

1. $\overline{y} = \tilde{y}$,
2. $\overline{y} = 9$,
3. $s^2 = 3.5$,
4. $\mathrm{range} = 6$.

A) 0 B) 1 C) 2 D) 3 E) 4

Question

Let $y = (1, 3, 4, 7, 11, 28)$. How many of the are following statements are true?

1. $\overline{y} = \tilde{y}$, ✗
2. $\overline{y} = 9$,
3. $s^2 = 3.5$,
4. $\mathrm{range} = 6$.

A) 0 B) 1 C) 2 D) 3 E) 4

Question

Let $y = (1, 3, 4, 7, 11, 28)$. How many of the are following statements are true?

1. $\overline{y} = \tilde{y}$, ✗
2. $\overline{y} = 9$, ✓
3. $s^2 = 3.5$,
4. $\mathrm{range} = 6$.

A) 0 B) 1 C) 2 D) 3 E) 4

Question

Let $y = (1, 3, 4, 7, 11, 28)$. How many of the are following statements are true?

1. $\overline{y} = \tilde{y}$, ✗
2. $\overline{y} = 9$, ✓
3. $s^2 = 3.5$, ✗
4. $\mathrm{range} = 6$.

A) 0 B) 1 C) 2 D) 3 E) 4

Question

Let $y = (1, 3, 4, 7, 11, 28)$. How many of the are following statements are true?

1. $\bar{y} = \tilde{y}$, ✗
2. $\bar{y} = 9$, ✓
3. $s^2 = 3.5$, ✗
4. $\mathrm{range} = 6$. ✗

A) 0 B) 1 C) *2* D) 3 E) 4

Quantiles

The *q*th quantile is the point where at least $q \cdot 100\%$ of the data values are at or below the value.

There are some special quantiles called quartiles (quarters of the data).

Q1 First quartile. 0.25 quantile.

Q2 Second quartile. 0.5 quantile. Median.

Q3 Third quartile. 0.75 quantile.

The interquartile range is the difference between Q3 and Q1.

$$\text{IQR} = Q3 - Q1$$

It contains the contains the center 50% of the data.

Percentiles, Quantiles and Quartiles

- **Quantiles** are the cut points that divide our observations into equal sized groups.
- There is one less quantile than the number of groups created.
  - i.e. splitting our data into 4 groups requires 3 cuts
- Common quantiles have special name, for example
  - **quartile** creates 4 groups
  - **decile** creates 10 groups
  - **percentiles** creates 100 groups.

Percentiles, Quantiles and Quartiles

Note the realtionship between the three concepts:

Quartile	Quantile	Percentile	
–	0 quantile	0 percentile	
1 quartile (Q_1)	0.25 quantile	25 percentile	
2 quartile (Q_2)	0.50 quantile	50 percentile	*(or median)*
3 quartile (Q_3)	0.75 quantile	75 percentile	
–	1 quantile	100 percentile	

Example

Let the data $y = \{1, 2, 3, 4, 5, 6\}$ and median $\tilde{y} = \frac{3+4}{2} = 3.5$.

Q1 and Q3 are then the medians of the two subsets of data when divided at the median:

$$y_1 = \{1, 2, 3\} \qquad y_2 = \{4, 5, 6\} \qquad Q1 = 2 \qquad Q3 = 5$$

Question

Let $y = \{0, 1, \ldots 100\}$. How many of the following are true?

1. The median is 50 and Q3 is 75.
2. Each integer y_i is the $y_i/100$th quantile. i.e. 4 is the 0.05th quantile.
3. For every data set, Q2 is strictly less than Q3.
4. If the data is reversed the quantile values remain unchanged.

A) 0 B) 1 C) 2 D) 3 E) 4

Question

Let $y = \{0, 1, \ldots 100\}$. Which of the following are true?

1. The median is 50 and Q3 is 75.
2. Each integer y_i is the $y_i/100$th quantile. i.e. 4 is the 0.05th quantile.
3. For every data set, Q2 is strictly less than Q3.
4. If the data is reversed the quantile values remain unchanged.

A) 0 B) 1 C) 2 D) 3 E) 4

Question

Let $y = \{0, 1, \dots 100\}$. Which of the following are true?

1. The median is 50 and Q3 is 75. ✓
2. Each integer y_i is the $y_i/100$th quantile. i.e. 4 is the 0.05th quantile.
3. For every data set, Q2 is strictly less than Q3.
4. If the data is reversed the quantile values remain unchanged.

A) 0 B) 1 C) 2 D) 3 E) 4

Question

Let $y = \{0, 1, \ldots 100\}$. Which of the following are true?

1. The median is 50 and Q3 is 75. ✓
2. Each integer y_i is the $y_i/100$th quantile. i.e. 4 is the 0.05th quantile. ✓
3. For every data set, Q2 is strictly less than Q3.
4. If the data is reversed the quantile values remain unchanged.

A) 0 B) 1 C) 2 D) 3 E) 4

Question

Let $y = \{0, 1, \ldots 100\}$. Which of the following are true?

1. The median is 50 and Q3 is 75. ✓
2. Each integer y_i is the $y_i/100$th quantile. i.e. 4 is the 0.05th quantile. ✓
3. For every data set, Q2 is strictly less than Q3. ✗
4. If the data is reversed the quantile values remain unchanged.

A) 0 B) 1 C) 2 D) 3 E) 4

Question

Let $y = \{0, 1, \ldots 100\}$. Which of the following are true?

1. The median is 50 and Q3 is 75. ✓
2. Each integer y_i is the $y_i/100$th quantile. i.e. 4 is the 0.05th quantile. ✓
3. For every data set, Q2 is strictly less than Q3. ✗

   If data is comprised of the same number repeated then Q2 = Q3.
4. If the data is reversed the quantile values remain unchanged.

A) 0 B) 1 C) 2 D) 3 E) 4

Question

Let $y = \{0, 1, \dots 100\}$. Which of the following are true?

1. The median is 50 and Q3 is 75. ✓
2. Each integer y_i is the $y_i/100$th quantile. i.e. 4 is the 0.05th quantile. ✓
3. For every data set, Q2 is strictly less than Q3. ✗

   If data is comprised of the same number repeated then Q2 = Q3.
4. If the data is reversed the quantile values remain unchanged. ✓

A) 0 B) 1 C) 2 D) *3* E) 4

Five Number Summary

A five number summary consists of the following in this order:

1. minimum,
2. Q1,
3. median,
4. Q3, and
5. maximum.

Example

Using $y = 1, 2, 3, 4, 5, 6$ the five number summary would be

```
> y <- c(1,2,3,4,5,6)
> fivesum(y)
[1] 1.0 2.0 3.5 5.0 6.0
```

Question

Which of the following are true?

1. Variance is always non negative.
2. Standard deviation can be zero.
3. If $a > b$ then $\mathrm{quantile}(a) \geq \mathrm{quantile}(b)$.
4. The five number summary provides the mean of the dataset.

A) 0 B) 1 C) 2 D) 3 E) 4

Question

Which of the following are true?

1. Variance is always non negative.
2. Standard deviation can be zero.
3. If $a > b$ then $\mathrm{quantile}(a) \geq \mathrm{quantile}(b)$.
4. The five number summary provides the mean of the dataset.

A) 0 B) 1 C) 2 D) 3 E) 4

Question

Which of the following are true?

1. Variance is always non negative. ✓
2. Standard deviation can be zero.
3. If $a > b$ then $\mathrm{quantile}(a) \geq \mathrm{quantile}(b)$.
4. The five number summary provides the mean of the dataset.

A) 0 B) 1 C) 2 D) 3 E) 4

Question

Which of the following are true?

1. Variance is always non negative. ✓
2. Standard deviation can be zero. ✓
3. If $a > b$ then $\mathrm{quantile}(a) \geq \mathrm{quantile}(b)$.
4. The five number summary provides the mean of the dataset.

A) 0 B) 1 C) 2 D) 3 E) 4

Question

Which of the following are true?

1. Variance is always non negative. ✓
2. Standard deviation can be zero. ✓
3. If $a > b$ then $\mathrm{quantile}(a) \geq \mathrm{quantile}(b)$. ✓
4. The five number summary provides the mean of the dataset.

A) 0 B) 1 C) 2 D) 3 E) 4

Question

Which of the following are true?

1. Variance is always non negative. ✓
2. Standard deviation can be zero. ✓
3. If $a > b$ then $\mathrm{quantile}(a) \geq \mathrm{quantile}(b)$. ✓
4. The five number summary provides the mean of the dataset. ✗

A) 0 B) 1 C) 2 D) 3 E) 4

Question

Which of the following are true?

1. Variance is always non negative. ✓
2. Standard deviation can be zero. ✓
3. If $a > b$ then $\mathrm{quantile}(a) \geq \mathrm{quantile}(b)$. ✓
4. The five number summary provides the mean of the dataset. ✗

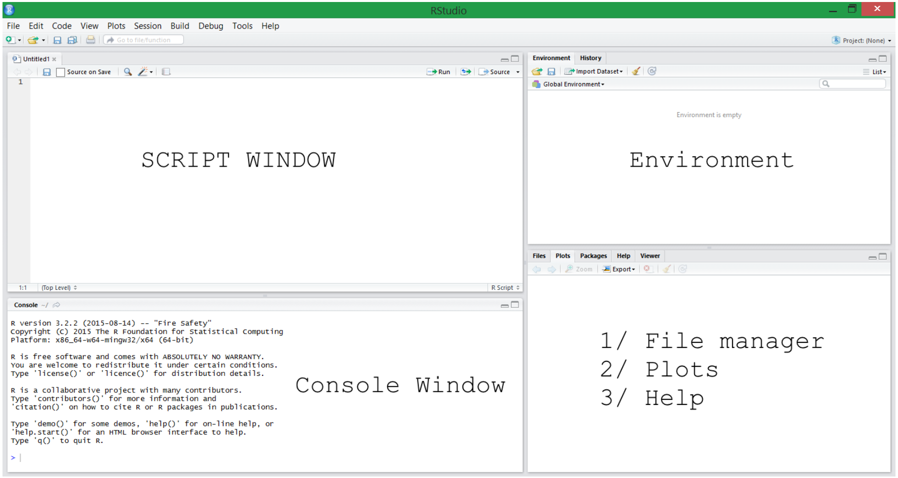
A) 0 B) 1 C) 2 D) ***3*** E) 4

Note the $\geq$ of 3.

RStudio

- While you could use the basic R graphical user interface (GUI) to interface with R, many people choose to run R through RStudio.
- RStudio is an integrated development environment (IDE) for R and—for the purpose of our course—will make it easy to write, run and orangize R code all in one window.
- Install R first! `cran.rstudio.com/`
- Download RStudio at: `rstudio.com/products/rstudio/download/`

RStudio Environment



```
R version 3.2.2 (2015-08-14) -- "Fire Safety"
Copyright (C) 2015 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

Background

- Rstudio has four panels (we can move and adjust their position in Preferences):
    1. Console panel
    2. Source editor
    3. Environment panel
    4. Files, Plots, and Help panel
- Let's start by going through these in a little bit more detail.

RStudio IDE

Console Where commands are executed. Where you see the input/output.

Script Window Draft and save code (only input) and run in the console by typing Ctrl / Cmnd (Windows/Mac) + Enter , or pressing Run .

Environment Shows saved variables and datasets

File Browser/Plots/Help Show files in working directory and generated plots, help, open window.

Console panel

- To see how we can use the Console with keyboard input, simply navigate to the panel and start tying commands:

```
2+4
print("Hello World!")
```

- Notice that R defaults functionality is displays output in the console directely beneith the inputs.
    - Shows input (blue), output (black) and any errors or warnings (red) (different “Themes" can be chosen to display different colours).

Sidenote: I will be using **knitr** to produce R lecture notes.

- R input appears in highlighted text (colour will depend on the object)
- comments in purple italics, and
- output is printed with two preceeding hashtags[1]. you will not see the two hashtags on standard output in R and Rstudio.

```
# comments in purple italics
 2 + 3 # input
## [1] 5
## ^ denotes output
```

[1]provides an easy to copy and paste R code

R scripts

- More commonly, we will be executing commands from an R script.
- An R **script** is a text document containing R commands and comments (i.e. all of your inputs without any output).
  - Hashtags # indicate the start of a comment.
  - A shortcut to commenting in Rstudio is shift + Command + C on a Mac and shift + Contrl + C on a Windows machine
- An R script ends with the extension `.R`
- Apart from one-off calculations, it is always a good idea to save an R script (e.g. easy to share and reproduce you results)

R scripts

A very simple R script (`script.R` on Canvas):

script.R

```
# Simple addition
2 + 3 # this is a comment
# Simple subtraction
2 - 3
# Simple multiplication
print(2*3)
# Simple power
2^3 # another comment
```

Running R scripts

- To run code from an R script, simply:
  1. open the file (which will have a .R extension) in RStudio[2]
  2. Place your cursor on the line you wish to execute and press Run or type Cmnd /Ctrl (Mac/Windows) + ENTER

- This will essentially copy and paste the text from the script window into your console window to be run.

- Upon executing a command, your cursor will automatically navigate to the next execuatble command.

[2]for the remainder of the course I will assume you are using RStudio

Running R scripts

- Sourcing a script is equivalent to running the entire contents of the file in your console.
- To source an R script either:
  - Open the .R file in Rstudio and click the Source button
  - Open the .R file and type Shift + Ctrl + S (Windows) or Shift + Cmnd + S if you are on a Mac.
  - Run the following command in your console:

```r
source("simple.R") # for example
```

  - Note that all lines of code were run, but only the values wraped in `print()` statements appear in the console window.

The `print` function will print to the standard output (console in RStudio)

```
print("Hello World!")

## [1] "Hello World!"
```

Print statements are often used in conjunction with `paste()` functions

```
x = 5

print(paste("x =", x))

## [1] "x = 5"

print(paste("x", x, sep=" = "))

## [1] "x = 5"

print("x=",x) # doesn't work like python

## [1] "x="
```

Working Directory

- Running the `source("simple.R")` command requires that `simple.R` is saved in your current working directory.
- The working directory is the “home base” of your R program.
- All files are written to and read from the working directory.
- N.B. If you want to read a file that is not in your working directory, you will have to specify the complete path, eg.

```
# tilde (~) is shortform for your home directory
source("~/Dropbox/Data301/08R/data/simple.R")
```

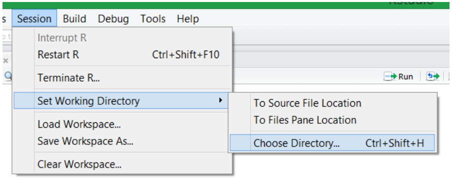
What is my home directory? (`/home/<username>` for Macs)

Working Directory

There are a few ways to set your working directory in RStudio:

1. Using the interface by doing

   `Session` $\rightarrow$ `Set Working Directory` $\rightarrow$ `Choose Directory`



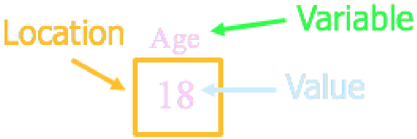
2. Using the `setwd()` command

```
setwd("c:/Documents/my/working/directory")
```

N.B. To check your working directory use `getwd()`.

Objects and Variables

- **Everything** you see or create in R is an object.
- This is an abstract term for anything that can be assignment to a varible.
- A variable is a name that refers to a location that stores a data



value.

IMPORTANT: The value at a location can change.

Some R Basics

- Like Python, R is case sensitive.
- Unlike Python, R in not that particular about white space and indent.

```
      x = 5 # both OK!
x = 7
```

- R commands may be separated either by a semi-colon or a newline (or both).

```
x = 7
y = 3 # or
x = 7; y = 3
x = 7; # ok too
```

Basics of R

We can either use the = or <- for the assignment operator. There is some debate on the preferred syntax

```
x <- 5; # both OK!
x = 5
```

Curly brackets { } are used to group commands together.

```
for (i in c(1,2,3)) {
  print(i)
  print(i + 1)
print(i - 1)  # indent doesn't matter
}
```

Basic Operations for Vectors

We can also perform basic operations on vectors:

Operation	Command
natural log	`log(x)`
Exponent	`exp(x)`
Log base 10	`log10(x)`
Absolute value	`abs(x)`
Square root	`sqrt(x)`
Sum	`sum(x)`
Number of elements in x	`length(x)`

Unique elements of x	`unique(x)`
Mean	`mean(x)`
Variance	`var(x)`
Standard Deviation	`sd(x)`
Minimum value	`min(x)`
Maximum value	`max(x)`
Smallest and largest values	`range(x)`
median	`median(x)`
Basic statistical summary	`summary(x)`
Sort	`sort(x)`

Question

How many of the following are true?

1. R is case-sensitive.
2. A command in R can be terminated by a semi-colon.
3. Indentation is the syntax used to group statements together.
4. A single line comment starts with #.

A) 0 B) 1 C) 2 D) 3 E) 4

Question

How many of the following are true?

1. R is case-sensitive.
2. A command in R can be terminated by a semi-colon.
3. Indentation is the syntax used to group statements together.
4. A single line comment starts with #.

A) 0 B) 1 C) 2 D) 3 E) 4

Question

How many of the following are true?

1. R is case-sensitive. ✓
2. A command in R can be terminated by a semi-colon.
3. Indentation is the syntax used to group statements together.
4. A single line comment starts with #.

A) 0　B) 1　C) 2　D) 3　E) 4

Question

How many of the following are true?

1. R is case-sensitive. ✓
2. A command in R can be terminated by a semi-colon. ✓
3. Indentation is the syntax used to group statements together.
4. A single line comment starts with #.

A) 0 B) 1 C) 2 D) 3 E) 4

Question

How many of the following are true?

1. R is case-sensitive. ✓
2. A command in R can be terminated by a semi-colon. ✓
3. Indentation is the syntax used to group statements together. ✗
4. A single line comment starts with #.

A) 0 B) 1 C) 2 D) 3 E) 4

Question

How many of the following are true?

1. R is case-sensitive. ✓
2. A command in R can be terminated by a semi-colon. ✓
3. Indentation is the syntax used to group statements together. ✗
4. A single line comment starts with #. ✓

A) 0 B) 1 C) 2 D) 3 E) 4

Question

How many of the following are true?

1. R is case-sensitive. ✓
2. A command in R can be terminated by a semi-colon. ✓
3. Indentation is the syntax used to group statements together. ✗
4. A single line comment starts with #. ✓

A) 0 B) 1 C) 2 D) *3* E) 4

Question

How many of the are following statements are true?

- In RStudio, R commands are executed in the console window
- R scripts store both R input and output
- We can run R without RStudio but we cannotrun Rstudio without R
- A working directory can not be changed once an R session is started

A) 0 B) 1 C) 2 D) 3 E) 4

Question

How many of the are following statements are true?

- In RStudio, R commands are executed in the console window
- R scripts store both R input and output
- We can run R without RStudio but we cannotrun Rstudio without R
- A working directory can not be changed once an R session is started

A) 0 B) 1 C) 2 D) 3 E) 4

Question

How many of the are following statements are true?

- In RStudio, R commands are executed in the console window ✓
- R scripts store both R input and output
- We can run R without RStudio but we cannotrun Rstudio without R

- A working directory can not be changed once an R session is started

A) 0 B) 1 C) 2 D) 3 E) 4

Question

How many of the are following statements are true?

- In RStudio, R commands are executed in the console window ✓
- R scripts store both R input and output ✗
- We can run R without RStudio but we cannotrun Rstudio without R
- A working directory can not be changed once an R session is started

A) 0 B) 1 C) 2 D) 3 E) 4

Question

How many of the are following statements are true?

- In RStudio, R commands are executed in the console window ✓
- R scripts store both R input and output ✗
- We can run R without RStudio but we cannotrun Rstudio without R ✓
- A working directory can not be changed once an R session is started

A) 0 B) 1 C) 2 D) 3 E) 4

Question

How many of the are following statements are true?

- In RStudio, R commands are executed in the console window ✓
- R scripts store both R input and output ✗
- We can run R without RStudio but we cannotrun Rstudio without R ✓
- A working directory can not be changed once an R session is started ✗

A) 0 B) 1 C) *2* D) 3 E) 4

Data types

- Here's a list of some important data types used in R:
  1. Integer (Whole Numbers)
  2. Numeric (Real Numbers)
  3. Character
  4. Logical (Boolean TRUE/FALSE)
  5. Factor
  6. Complex (we won't focus on this type)
- As with Python, variables of certain types need not be declared, and `R` will do its best at determining the most appropriate data type.
- We can check the type of an object using `class()`

```r
# class works like type() in Python
class(1)   # like Python float type

## [1] "numeric"

class(as.integer(1)) # like Python int type

## [1] "integer"

z = 1 + 2i; class(z)

## [1] "complex"

class(TRUE) # notice all in caps!  (True in Python)

## [1] "logical"

class("irene") # like Python string

## [1] "character"
```

Numeric and Integer

- Often the default storage of numbers in R is in as numeric.
- To force a variable to be an interger, we need to use the `as.integer()` function, or use the “L” suffix.

```
numobj = 2 # numeric (not integer)
as.integer(numobj) # integer
## [1] 2
intobj = 2L # integer
as.integer('4') # like int() in Python
## [1] 4
```

N.B. forcing data to be a certain type is called data coercion. See slide 63

Character

- A single letter or string of characters will be recorded as a `character` object.
- You can use double quotes or single quotes for all of these specifications (just be consistent)

```
# note our usage of quotes
charobj = "s"
strobj = 'string'
numstr = as.character(4) # like str(4) in Python
# badobj = "str' # unmatched quotes = bad
```

Aside

Auto-complete

Notice that R trys to help us out by autocompleting our quotes (i.e. when we type one quotation/parenthesis the closing one will appear automatically. If you don't like this feature, you can turn it off (but I suggest you keep it!)

Logical

In R, `T` and `F` are reserved for logicals (i.e. True/False objects) and R reads `T` the same as `TRUE` or `F` the same as `FALSE`.

```
x = c(T, FALSE,TRUE, F) # notice we don't use quotes
x

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

Oftentimes, True and False conditions are coded as 0 and 1s. To covert them to logical, use `as.logical()`.

```
as.logical(0)

## [1] FALSE

as.logical(1)

## [1] TRUE
```

Factor

- If you want a variable to be a factor type (so that different symbols are regarded as a level), use `var=factor(var)`.

```
notfactor = c("H","M", "M", "L", "H")
notfactor

## [1] "H" "M" "M" "L" "H"

(fvec = factor(notfactor))

## [1] H M M L H
## Levels: H L M

levels(fvec) # to see the levels

## [1] "H" "L" "M"
```

Factor

- We also have the option to specify the labels to something more meaningful:

```
# make sure your labels appear in the same order as levels(fvec)
fvec = factor(x=notfactor, labels=c("High","Low","Medium"))
fvec

## [1] High   Medium Medium Low    High
## Levels: High Low Medium
```

Ordered Factors

Sometimes the order of the factors do not matter (eg. eye colour). If however we want to specify some meaningful order, we could use levels. In this example, since `L` (low) $<$ `M` (medium) $<$ `H` (high), we could specify ordered levels using `ordered = TRUE`:

```
(fvec = factor(x=notfactor, levels=c("L","M","H")))

## [1] H M M L H

## Levels: L M H

(fvec2 = factor(x=notfactor, levels=c("L","M","H"), ordered = TRUE))

## [1] H M M L H

## Levels: L < M < H
```

Ordered Factors

Notice how we can compare the levels in `fvec2` using relational operators whereas in `fvec` we cannot:

```
fvec[1]>fvec[3]

## Warning in Ops.factor(fvec[1], fvec[3]):  '>' not meaningful for
factors

## [1] NA

fvec2[1]>fvec2[3]

## [1] TRUE

print(paste(fvec2[1], "is treated as greater than", fvec[3]))

## [1] "H is treated as greater than M"
```

Ordered Factors

We could use this in conjunction with `labels`:

```
fvec3 = factor(x=notfactor, levels=c("L","M","H"), labels=c("Low","Medium"
fvec3
## [1] High   Medium Medium Low    High
## Levels: Low < Medium < High
```

Factor

- The case my also arise that we need to specify levels not seen in the data set. In that case we can specify the `levels` argument:
- Using `letters` a built in vector containing the 26 letters of the alphabet;

```
x = c("o","q","h","n","s","b","u","d","p","r")
xlist = factor(x, levels=letters)
table(xlist)
```

```
## xlist
## a b c d e f g h i j k l m n o p q r s t u v w x y z
## 0 1 0 1 0 0 0 1 0 0 0 0 0 1 1 1 1 1 1 0 1 0 0 0 0 0
```

Aside

Built-in Constants

- On the previous slide we used the built-in constant `letters`
- Other build in constants include:
  - `LETTERS` (capital letters)
  - `month.name` (eg. "January", "February", etc..)
  - `month.abb` (eg. "Jan", "Feb", etc..)
  - `pi` 3.141593

Try it: in R

Question

In an R program

1. Make a comment with your name and student number.
2. Calculate the following $4 \times 5 - 12^3$, $\mathrm{e}^{3\times4}$, $\sin(4\pi - 6)$.
3. Guess then check with `class` the types of the following variables

```r
var1 <- FALSE
var2 <- T
var3 <- "TRUE"
var4 <- 3^4 - 10
var5 <- "Hello World!"
```

Data structures

- R has a wide variety of data structures including:
  - scalars, eg

```
scalar1 <- 1
scalar2 <- "string"
```

  - vectors (numerical, character, logical)

```
vec1 <- c(1,2,3); vec2 <- c("one", "two", "three")
vec3 <- c(TRUE, FALSE, T)
```

  - matrices
  - data frames, and
  - lists

Vectors

- A vector is the most basic data structure in R and can be of two types:
  - atomic vectors
  - lists
- As we did in the examples above, you can create a vector using `c` (see `?c`[3] for more details on this *combine* function)

[3] you can get help on any function running `?functionname`

Vectors

Atomic Vectors can be either vectors characters, logical, integers or numeric (but can not mix types)

```
# example of a numeric vector:
y=c(2,3,0,3,1,0,0,1)
# example of a character vector:
letters<-c('A','B','C')
# example of a logical vector:
lvec <- c(TRUE, FALSE, FALSE)
```

Unlike Python lists, we cannot mix types within vectors:

```
# the integer 1 gets converted to character
print(vec <- c(1, "a", "b"))

## [1] "1" "a" "b"
```

coercion

- By *atomic vectors* we mean they can only take on one data type.
- If we specify more than one type in a single vector, R will convert the mixed types to a single type which it deems most appropriate.
- The coercion will move towards the one that's easiest to coerce to.
- You can coerce vectors explicitly using the `as.`<*type*> (eg. `as.numeric`, `as.character`, etc.)

```
(x = c("apple", 2, TRUE)) # converts all to character

## [1] "apple" "2"     "TRUE"

(x = c(TRUE, 4)) # converts all to numeric

## [1] 1 4

(x = c(2L, -1.3)) # converts to numeric

## [1]  2.0 -1.3

(x = c(TRUE, 0, 1, FALSE)) # converts all to numeric

## [1] 1 0 1 0

as.logical(x = c(TRUE, 0, 1, FALSE)) # converts to logical

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

Vectors

Numeric vectors can be created a number of different ways.

```
# Ceate a vector of size 10, where each element is a 2:
y = rep(2,10)
# This specifies a sequence of integers:
y = 3:12
#This specifies a sequence of real numbers:
z = seq(from=3,to=12,by=1) # by = 1 is the default
#This specifies a sequence of real numbers:
z = seq(from=2,to=1,by=-0.1)
```

The end in `seq()` is inclusive

`seq()` is like the Python `range(start, end, step)` function, only now, `end` **is** inclusive!

Aside

Notation

- We say that `seq()` is a **function** for which we have specified three **arguments** (having argument names: `from`, `to` and `by`).
- Assuming we are putting the values in the correct order, you do not need to specify the argument names explicitly.

```r
seq(1,2,0.1) # same as seq(from=1,to=2,by=0.1)

##  [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0

# if we want to specify arguments out of order,
# we HAVE to include argument names:
seq(by=0.1, to=2, from=1) # arguments in green

##  [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
```

Vectors

Here are some ways you can generate “random" data in R:

```
# Ceates a vector of size 10, of random numbers between 0 and 1
y = runif(10)
# Creates a vector of size 3 sampled from x without replacement
(y = sample(x=c(1,2,3), size=3))

## [1] 1 3 2

(y = sample(x=c(1,2,3), size=3)) # random output each time

## [1] 3 2 1

# Creates a vector of size 3 sampled from x with replacement
(y = sample(x=c(1,2,3), size=3, replace=TRUE))

## [1] 3 2 3
```

Vectors

We can also create empty vectors

```r
x <- vector() # defaults to logical (FALSE)
x <- vector(length = 10) # defaults to logical vec of FALSES
x <- vector("character", length = 10)
x <- vector("numeric",   length = 10)
x <- vector("integer",   length = 10)
x <- vector("logical",   length = 10)
```

Notice the difference in creating empty vectors in R from Python. For instance, the following will produce an error

```r
x <- ()
## Error:  <text>:1:7:  unexpected ')'
## 1:  x <- ()
```

Scalars

Scalars are just vectors of length 1. To check the length of any vector use `length()`

```
x = 4; length(x)

## [1] 1

y=c(2,3,0,3,1,0,0,1); length(y)

## [1] 8

# notice that this does not count characters in a string:
length("Hello") # see nchar("Hello") for # of characters

## [1] 1
```

`length()` in R instead of `len()` in Python

`length` solely provides the length of a vector/list.

Indexing Vectors

To index an element from a vector, use single square brackets

```
z = c("apples","bananas","oranges", "pineapples"); z[1]

## [1] "apples"

z[2:3]

## [1] "bananas" "oranges"

z[c(4,2)]

## [1] "pineapples" "bananas"

z[-1] # removes the first item in the vector

## [1] "bananas"    "oranges"    "pineapples"
```

Notice that the index begins at 1 NOT 0!!!!!!

Also notice the difference in the `[-1]` notation.

We have the option of including names for our vector elements (similar to a Python dictionary)

```
z = c(123456, 25, 2019); names(z) <- c("studentID","age","year")
z["studentID"] # same as z[1]

## studentID
##    123456

z[c("age, year")] # wont work like z[c(2,3)]

## <NA>
##   NA

z[c(FALSE, TRUE, TRUE)] # index using logicals (same as z[c(1,2)])

##  age year
##   25 2019

z[-"year"] # wont work like z[-3]

## Error in -"year":  invalid argument to unary operator
```

To add a new element/delete/replace an element:

```
z = c("first","second","third", "fourth")
# adds a fith element and replaces the second
z[5] = "fifth";  z[2] = "2nd"
z
## [1] "first"  "2nd"    "third"  "fourth" "fifth"
#removes the second element
(z = z[-2])
## [1] "first"  "third"  "fourth" "fifth"
```

R does not use methods like the `.append()` as Python did.

Notice what happens if we add a 10th element to z (without specifying elements 6–9):

```
z[10] = "tenth"
z
```

```
##  [1] "first"  "third"  "fourth" "fifth"  NA       NA       NA
##  [8] NA       NA       "tenth"
```

We can also add new named elements

```
z = c(123456, 25, 2019); names(z) <- c("studentID","age","year")
# adds a forth element with the students major
z["major"] = "COSC"
z

## studentID       age      year     major
##  "123456"      "25"    "2019"    "COSC"
```

Notice how the addition of this forth element cuase the vector to change to a vector of characters (because of data coercion!)

Vector Operations

- Basic operations (like the ones outlined on slide 40) can be a applied to vectors.
- Like the map function and numpy arrays in Python, these functions will be applied *elementwise*.

```
(y= sample(1:20, 9)) # samples 9 random numbers from 1 to 20

## [1] 15 19 14  3 10  2  6 11  5

2 * y # multiplies each element by 2

## [1] 30 38 28  6 20  4 12 22 10

y + y # equivalent

## [1] 30 38 28  6 20  4 12 22 10
```

Vector Operations

- Notice that operations like `sort()` do not change the input vector unless that vector is reassigned.

```
print(y)

## [1] 15 19 14  3 10  2  6 11  5

sort(y)

## [1]  2  3  5  6 10 11 14 15 19

print(y)

## [1] 15 19 14  3 10  2  6 11  5

# this time reassign it
y = sort(y, decreasing = TRUE)
print(y)

## [1] 19 15 14 11 10  6  5  3  2
```

Matrices

- **Matrices** can be viewed as column vectors stacked side by side or row vectors stack one on top of another
- In R, matrices are really just a multi-dimensional vector (similar to numpy arrays in Python)
- The main difference between matrices and the vectors discussed in the previous slides is that a matrix has a **dimension** (similar the shape of an numpy array in Python).
- The dimenions can be found using the `dim()` function.

Matrices

```
(m <- matrix(1:8, nrow = 2, ncol = 4))

##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    2    4    6    8

dim(m) # gives the number or rows and columns

## [1] 2 4

class(m)

## [1] "matrix"
```

Matrices

- By default, matrices are constructed columnwise.
- You can always change to row-wise by specifying `byrow=TRUE`

```
(m <- matrix(1:6, nrow=2, ncol =3)) # fill columnwise

##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6

(m <- matrix(1:6, nrow=2, ncol =3, byrow=TRUE)) #fill row-wise

##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

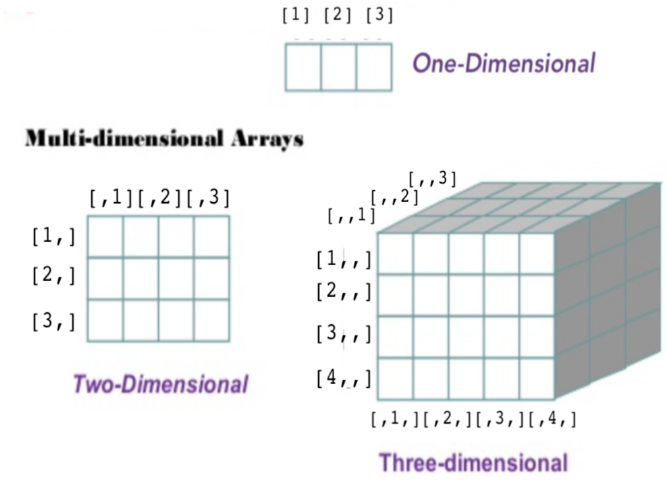
Matrices

An alternative way of constructing matrices is to use `array()`

```
array(1:6, dim=c(2,3)) # a 2 x 3 array  (2 dimensional)

##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6

# returns 2 matrices of dimension 4 x 3
(arr3d <-  array(1:24, dim=c(4,3,2))) # a 4 x 3 x 2 array (3 dimensional)

## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
```

Array visualization

Image adapted from (image source)



Named Matrices

We may also choose to name our rows and columns

```
rownames(m) = c("row1", "row2")
colnames(m) = c("col1", "col2", "col3")
m
```

```
##      col1 col2 col3
## row1    1    2    3
## row2    4    5    6
```

Indexing

- To index an element from a matrix, we can again use square brackets, but we need to specify TWO numbers `[row, col]`.
- To index an element from a higher dimensional array, we use square brackets with as many indeces as there are dimensions `[row, col, dept]` for 3-dimensional arrays.
- If we want to extract an entire row, we'll leave the `col` field blank (indicating we want all colums).
- If we want to extract an entire column, we'll leave the `row` field blank (indicating we want all rows).

Indexing

The following extracts a column, row, and cell from matrix `m`,

```
m[,3] # extract the third column of m

## row1 row2

##    3    6

m[1,] # extract the first row of m

## col1 col2 col3

##    1    2    3

m[1,3] # extract the element in the 1st row and 3rd column

## [1] 3

arr3d[4,3,2] # 4th row and 3rd column from the 2nd matrix

## [1] 24
```

Alternatively we could call them by name

```
# extract the third column of m
m[,"col3"] # same as m[,3]

## row1 row2
##    3    6

# extract the first row of m
m["row1",] # same as m[1,]

## col1 col2 col3
##    1    2    3

# extract the element in the 1st row and 3rd column
m["row1","col3"] # same as m[1,3]

## [1] 3
```

Indexing

Note that we can also use one number (tricker this way)

```
(m = matrix(11:4, 2,4))

##      [,1] [,2] [,3] [,4]
## [1,]   11    9    7    5
## [2,]   10    8    6    4

m[8] # extracts the 8th element in m

## [1] 4

(mVector = as.numeric(m))

## [1] 11 10  9  8  7  6  5  4

mVector[8]

## [1] 4
```

Indexing

Like the vectors discussed earlier, matices fall under the atomic vector category and therefore can only store data of one type:

```
m = matrix(c("bananas", 2, TRUE, 0),  nrow=2, ncol=2)
# notice how all coerce to characters:
m
##      [,1]      [,2]
## [1,] "bananas" "TRUE"
## [2,] "2"       "0"
```

lists

- If we want our vector to contain objects of different data types we need to create a list.
- We can think of lists as a special type of vector that can mix data types and structures
  - For example, an element in a list can contain yet another list (i.e nested lists)
- These are very similar to Python lists but remember R lists start with the index 1 and Python lists start with the index 0.

Notice how lists do not force all elements to be of the same type (as was the case with atomic vectors)

```
(x = list("apple", 2, TRUE))

## [[1]]
## [1] "apple"
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] TRUE

(y = c("apple",2,TRUE))

## [1] "apple" "2"     "TRUE"
```

lists

Unlike vectors, the member of a list are accessed using double square brackets:

```
x[[1]] # first member

## [1] "apple"

x[[2]] # second member, and so on ...

## [1] 2
```

lists

(my) notation

- While I have specified each member to be a single word/number/logical, we could also have specified them to be vectors or matrices or another list!

- In an attempt to avoid confusion, I will use *member*s to refer to the elements of the list and *elements* to refer to elements within the vectors/matrices.

lists

Indexing

- Members will be referenced using double square brackets `[[]]`
- We will reference the elements within each member using square brackets:
  - single square brackets if its a vector `[]`
  - double square brackets if its a list `[[]]`
- Notice how R outputs uses the same square bracket referencing in the standard output.

```r
(y = list(c("apples", "banana", "four"), list(1:3,"hi"), matrix(1:6,2)))
## [[1]]
## [1] "apples" "banana" "four"
##
## [[2]]
## [[2]][[1]]
## [1] 1 2 3
##
## [[2]][[2]]
## [1] "hi"
##
##
## [[3]]
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

The first memeber of y is a vector. We therefore reference its elements by a single index in square brackets

```
y[[1]] # this is a vector

## [1] "apples" "banana" "four"

# the third element of this vector is obtained in the usual way
y[[1]][3]

## [1] "four"
```

If it helps, you can think of `y[[1]]` as a new vector object.

```
(new = y[[1]]) # this is a vector

## [1] "apples" "banana" "four"

new[1]

## [1] "apples"
```

The second memeber of `y` is *another* list. We therefore reference its elements by double square brackets

```
y[[2]]
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] "hi"
```

If it helps, you can think of `y[[2]]` as a new list object

```
newlist = y[[2]] # this is a list
newvec = newlist[[1]][1] # this is a vector
```

The first memeber of `y[[2]]` is a vector. We therefore reference its elements using single square brackets.

```
y[[2]] # this is a list

## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] "hi"

y[[2]][[1]] # this is a vector

## [1] 1 2 3

y[[2]][[1]][3] # this is a scalar

## [1] 3
```

The second memeber of `y[[2]]` is a scalar (ie a vector of length 1). We therefore dont need anything else to reference its single and only element

```
y[[2]] # this is a list

## [[1]]

## [1] 1 2 3

##

## [[2]]

## [1] "hi"

y[[2]][[2]] # this is a scalar (one length vector)

## [1] "hi"
```

Named lists

Alternatively we could have names for our list members

```
numbers= c(2,4) # can define the member outside first
y = list(words=c("apples", "banana", "four"),
         numbers, logicals=TRUE)
y$words # same this as y[[1]]

## [1] "apples" "banana" "four"

y$numbers # NULL since the name "numbers" is not saved

## NULL

y$logicals # same this as y[[3]]

## [1] TRUE
```

Named lists

Notice that the object name of `numbers` was not perserved in the naming of list memebers:

```
names(y)

## [1] "words"    ""         "logicals"
```

To name the second member in the list `numbers` we would have to explicitly name it using:

```
y = list(words=c("apples", "banana", "four"),
         numbers=numbers, logicals=TRUE)
y$numbers # the same as y[[2]]

## [1] 2 4
```

We could also name the list members after they have been assigned

```r
y = list(c("apples", "banana", "four"),
         c(2,4), TRUE)
names(y) <- c("words","numbers","logicals")
y
## $words
## [1] "apples" "banana" "four"
##
## $numbers
## [1] 2 4
##
## $logicals
## [1] TRUE
```

lists

We can index members of lists using:

- square brackets with either integer index numbers OR their member names (in quotations).
- or the dollar sign with the member names (not in quotations)

```
y$words
y[["words"]]
y[[3]]
```

The following are invalid:

```
y$3 # dollar sign wont work with number index
y[[words]] # double square brackets need names in quotes
```

lists

We can modify its content/add new content directly either using `[[]]` with numbers or $ with member names.

```r
y$words[3] = "oranges"#replace the 3rd element of the 1st member
y[[1]][4] = "pineapples"#add a 4th element to the 1st member
y[[4]] = c(0,1,1,0) # add a forth member to y
y$"fifth" = 3:12 # add a fifth member to y (named "fifth")
```

Data Frames

- Data frames are perhaps the most used data structure used in statistics.
- Like list, they can store different data types.
- Futhermore, they can contain additional attributes such as column and row names.
- When combining vectors in a data frame all must have the same size (i.e. length).
- Missing observations will be recorded as `NA`.

Data Frames

Here is an example of how to create a data frame and add to it:

```
Person=c('John', 'Jill', 'Jack')
Grade=c(45,92,91)
(Lab=data.frame(Person, Grade))
```

```
##   Person Grade
## 1   John    45
## 2   Jill    92
## 3   Jack    91
```

Data Frames

Here `Person` is longer than `Grade` and will produce an error:

```r
Person=c('John', 'Jill', 'Jack','James')
Grade=c(45,92,91)
(Lab=data.frame(Person, Grade))
## Error in data.frame(Person, Grade):  arguments imply differing
number of rows:  4, 3
```

The following is OK

```r
Person=c('John', 'Jill', 'Jack','James')
Grade=c(45,92,91, NA)
Lab=data.frame(Person, Grade)
```

Data Frames

Like matrices, we can create column and row names

```
colnames(Lab) # adopted from the variable names inputed

## [1] "Person" "Grade"

rownames(Lab) # default to increasing integers

## [1] "1" "2" "3" "4"

# we could also add row names if we choose
rownames(Lab) = c("Student1", "Student2", "Student3", "Student4")
Lab

##          Person Grade
## Student1   John    45
## Student2   Jill    92
## Student3   Jack    91
## Student4  James    NA
```

Data Frames

Like matrices, we index using single square brackets with two integer/name indeces (first number for row, second number for column)

```
Lab["Student1", c("Grade","Person")]

##          Grade Person
## Student1    45   John

# sames as
Lab[1,2:1]

##          Grade Person
## Student1    45   John
```

Data Frames

Unlike matrices, we can extract a column of this data.frame we using the $ notation:

```
Lab$Person

## [1] John  Jill  Jack  James

## Levels: Jack James Jill John

Lab$Grade

## [1] 45 92 91 NA

# N.B. This does not work for rows:

Lab$Student1

## NULL
```

Aside

Smart Autofill

- In some situations, R will try to autocomplete commands for us. When you begin to type `Lab$` for instance, you will see the options for the column names appear.
- You can use the arrow keys on your keyboard to navigate through the options.
- Press ENTER when you land on the option you want.

Data Frames

Adding a column to the data frame is as easy as typing:

```
Lab$Passed=c(FALSE,TRUE,TRUE,NA)
Lab
```

```
##          Person Grade Passed
## Student1   John    45  FALSE
## Student2   Jill    92   TRUE
## Student3   Jack    91   TRUE
## Student4  James    NA     NA
```

Attributes

- Each of these objects have so-called attributes.
- Here are some examples of attributes:
  1. names, dimension names
  2. dimensions
  3. class
  4. length

Attributes

```
attributes(Lab) # data vector defined in previous example

## $names
## [1] "Person" "Grade"  "Passed"
##
## $row.names
## [1] "Student1" "Student2" "Student3" "Student4"
##
## $class
## [1] "data.frame"

attributes(m) # matrix defined previously

## $dim
## [1] 2 2
```

Vectors in R

Question

How many of the following statements are true?

1. Vectors in R are indexed from 0.
2. `1:10` creates a vector of ten numbers.
3. A vector may have elements of different data types.
4. If `data <- 1:5` then `data[-1]` returns 5.

A) 0 B) 1 C) 2 D) 3 E) 4

Question

How many of the following statements are true?

1. Vectors in R are indexed from 0.
2. `1:10` creates a vector of ten numbers.
3. A vector may have elements of different data types.
4. If `data <- 1:5` then `data[-1]` returns 5.

A) 0 B) 1 C) 2 D) 3 E) 4

Question

How many of the following statements are true?

1. Vectors in R are indexed from 0. ✗
2. `1:10` creates a vector of ten numbers.
3. A vector may have elements of different data types.
4. If `data <- 1:5` then `data[-1]` returns 5.

A) 0 B) 1 C) 2 D) 3 E) 4

Question

How many of the following statements are true?

1. Vectors in R are indexed from 0. ✗
2. `1:10` creates a vector of ten numbers. ✓
3. A vector may have elements of different data types.
4. If `data <- 1:5` then `data[-1]` returns 5.

A) 0 B) 1 C) 2 D) 3 E) 4

Question

How many of the following statements are true?

1. Vectors in R are indexed from 0. ✗
2. `1:10` creates a vector of ten numbers. ✓
3. A vector may have elements of different data types. ✗
4. If `data <- 1:5` then `data[-1]` returns 5.

A) 0 B) 1 C) 2 D) 3 E) 4

Question

How many of the following statements are true?

1. Vectors in R are indexed from 0. ✗
2. `1:10` creates a vector of ten numbers. ✓
3. A vector may have elements of different data types. ✗
4. If `data <- 1:5` then `data[-1]` returns 5. ✗

A) 0 B) *1* C) 2 D) 3 E) 4

Lists and Matrices in R

Question

How many of the following statements are true?

1. Data values in a list may be of different types.
2. In a matrix, the number of rows and columns must be the same.
3. Given matrix `m`, `m[2]` would return an error.
4. Given matrix `m`, `m[,2]` would return all data in column 2.

A) 0 B) 1 C) 2 D) 3 E) 4

Lists and Matrices in R

Question

How many of the following statements are true?

1. Elements in a list may be of different types.
2. In a matrix, the number of rows and columns must be the same.
3. Given matrix `m`, `m[2]` would return an error.
4. Given matrix `m`, `m[,2]` would return all data in column 2.

A) 0 B) 1 C) 2 D) 3 E) 4

Lists and Matrices in R

Question

How many of the following statements are true?

1. Elements in a list may be of different types. ✓
2. In a matrix, the number of rows and columns must be the same.
3. Given matrix `m`, `m[2]` would return an error.
4. Given matrix `m`, `m[,2]` would return all data in column 2.

A) 0 B) 1 C) 2 D) 3 E) 4

Lists and Matrices in R

Question

How many of the following statements are true?

1. Elements in a list may be of different types. ✓
2. In a matrix, the number of rows and columns must be the same. ✗
3. Given matrix `m`, `m[2]` would return an error.
4. Given matrix `m`, `m[,2]` would return all data in column 2.

A) 0 B) 1 C) 2 D) 3 E) 4

Lists and Matrices in R

Question

How many of the following statements are true?

1. Elements in a list may be of different types. ✓
2. In a matrix, the number of rows and columns must be the same. ✗
3. Given matrix `m`, `m[2]` would return an error. ✗
4. Given matrix `m`, `m[,2]` would return all data in column 2.

A) 0 B) 1 C) 2 D) 3 E) 4

Lists and Matrices in R

Question

How many of the following statements are true?

1. Elements in a list may be of different types. ✓
2. In a matrix, the number of rows and columns must be the same. ✗
3. Given matrix `m`, `m[2]` would return an error. ✗
4. Given matrix `m`, `m[,2]` would return all data in column 2. ✓

A) 0 B) 1 C) *2* D) 3 E) 4

Question

Create a list called `grades` and add the following elements

1. Name (first name and last name),
2. Student number,
3. Assignment grades (multiple entries), and
4. Midterm grade.

Question

How many of the following are true?

1. Matrices are indexed using double square brackets `[[]]`
2. A matrix is another word for a two dimensional array
3. A scalar is another word for a vector of length 1
4. Columns in a data frame can be of varying length

A) 0 B) 1 C) 2 D) 3 E) 4

Question

How many of the following are true?

1. Matrices are indexed using double square brackets `[[]]`.
2. A matrix is another word for a two dimensional array
3. A scalar is another word for a vector of length 1.
4. Columns in a data frame can be of varying length.

A) 0 B) 1 C) 2 D) 3 E) 4

Question

How many of the following are true?

1. Matrices are indexed using double square brackets `[[]]`. ✗
2. A matrix is another word for a two dimensional array
3. A scalar is another word for a vector of length 1.
4. Columns in a data frame can be of varying length.

A) 0 B) 1 C) 2 D) 3 E) 4

Question

How many of the following are true?

1. Matrices are indexed using double square brackets `[[]]`. ✗
2. A matrix is another word for a two dimensional array ✓
3. A scalar is another word for a vector of length 1.
4. Columns in a data frame can be of varying length.

A) 0 B) 1 C) 2 D) 3 E) 4

Question

How many of the following are true?

1. Matrices are indexed using double square brackets `[[]]`. ✗
2. A matrix is another word for a two dimensional array ✓
3. A scalar is another word for a vector of length 1.✓
4. Columns in a data frame can be of varying length.

A) 0 B) 1 C) 2 D) 3 E) 4

Question

How many of the following are true?

1. Matrices are indexed using double square brackets `[[]]`. ✗
2. A matrix is another word for a two dimensional array ✓
3. A scalar is another word for a vector of length 1.✓
4. Columns in a data frame can be of varying length. ✗

A) 0 B) 1 C) *2* D) 3 E) 4

Data Importing

- Data entry can be either done by hand (impractical), or loaded from an existing file.
- `read.table` is a convenient way of reading data into R and storing it to a data frame.
- The data should be formated such that each rows contains information regarding one subject with data separated by white space/commas/tabs/.
- The first line may (or may not) contain a `header` with the names of the variables in each column.

Data Entry

Let’s take the data `intake.txt` data available on CANVAS which looks like this:

“pre"	“post"
5260	3910
5470	4220
5640	3885
6180	5160
6390	5645
6515	4680
6805	5265
7515	5975

Data Entry

To read the data in:

```
# the intake file is in a folder called data in my working directory
read.table("data/intake.txt", header=TRUE)

##     pre post
## 1  5260 3910
## 2  5470 4220
## 3  5640 3885
## 4  6180 5160
## 5  6390 5645
## 6  6515 4680
## 7  6805 5265
## 8  7515 5975
## 9  7515 6790
## 10 8230 6900
```

Data Entry

- `header=TRUE` indicates that the first line of file contains the names of the variables (rather than data values).
  - `header=TRUE` allows `pre` and `post` to be read in as column names.
- (`header=FALSE` by default)

Data Entry

Since we didn't save it to a variable, it simple prints it to the screen. This wont be very useful in practice, so save it to a variable with a name that makes sense.

```
intake = read.table("data/intake.txt", header=TRUE)
```

Aside

Name Rules

There are rules on the types of names we can give variables:

- No spaces
- No special characters (apart from periods and underscores)
- Can't begin with a number (but numbers can be used in the names elsewhere)
- See Google's R Style Guide for other `R` coding practices and naming conventions.

Data Entry

If we instead want to import the `athlete.csv` file available on CANVAS, we simply specify that our separator is a comma.

```
data = read.table("data/athlete.csv", sep=",", header=TRUE)
```

Alternatively, we could have used the `read.csv()` function

```
data = read.csv("data/athlete.csv") # header=TRUE by default in read.csv
```

If you want to specify a file that is outside of your working directory, simply specify the path name preceeding the filename. For example:

```
data <- read.csv("/Users/ivrbik/DATA103/data.csv", header=T)
```

Data Exporting

There is also a function that will allow you to write the contents of a data frame to a file. For instance, we could save our simple `Lab` data frame to a textfile called `Lab.txt` or `Lab.csv` using the following:

```
write.table(Lab, file="Lab.txt")
write.csv(Lab, file="Lab.csv")
```

Like the `read` functions, `write` will, by default, save these files to your working direction. To specify otherwise use:

```
write.table(Lab, file="<path>/Lab.txt")
```

Question

1. Create a data frame `mydata` with the following column names/data:
   1.1 `id` – numbers one to 5.
   1.2 `location` – “BC”, “BC”, “AB”, “MB”, “BC”.
   1.3 `value` – 10, 20, 30, 40, 50.
   1.4 Make location a factor.
2. Add one more column to your data frame that is a factor
   2.1 `success` – “Y”, “N”, ‘N”, ‘N”, “Y”.
3. Export it as as a csv file to a file called `mydata.csv` (you mave save it to your current working directory)

Tip of the Day

Tip:

To obtain a previous calculation in R, navigate to the console, and press the Up↑ /Down↓ key on your keyboard. Alternatively, you could access your previously executed commands from the History tab in the Environment panel in RStudio.