在我電腦上啟動 R: 打開 terminal → Type R.

The video time for this lesson is 35 minutes.

The recommended reading for this lesson is:
R Programming

https://s3.amazonaws.com/content.udacity-data.com/courses/gt-cse6242/recommended+reading/R.pdf

This course will use the R programming language for projects and assignments. The student may download this open source software package at:

The R Project for Statistical Computing
https://www.r-project.org/

Brent Wagenseller, a GT OMSCS student posted these instructions in Piazza (August. 2016) for anyone who might be using Ubutu to run R.

I initially installed R a little less than a year ago and it seems the instructions have changed slightly (I dont know if you still need to add the signed keys, it was required last year but may not be now). Also, you need to pick a mirror - I picked 'lib.stat.cmu.edu/R/CRAN/', but pick one from the mirror list on https://cran.r-project.org.

Not sure if anyone else is using Ubuntu as their main machine, but if you are - here are my saved (and greatly shortened) instructions.

Installation Instructions for R: https://cran.r-project.org

To install R on Ubuntu, switch to root and edit sources.list:

```
vi /etc/apt/sources.list
```

Now pick a mirror and add this to the bottom of sources.list (in this case I picked mirror 'lib.stat.cmu.edu/R/CRAN/':

```
## Packages for R

deb http://lib.stat.cmu.edu/R/CRAN/bin/linux/ubuntu xenial/
```

Note that the version can change with the version of Ubuntu changing; you can just go to the target and see the versions available (for example I navigated to http://lib.stat.cmu.edu/R/CRAN/bin/linux/ubuntu and saw that xenial – the latest for Ubuntu 16.04 – was available)

Save and exit sources.list by pressing the 'ESC' key and typing:

```
:wq!
```

The above will require specific signed keys to be added to your system; according to the R instructions, run these commands:

```
gpg --keyserver keyserver.ubuntu.com --recv-key E084DAB9
gpg -a --export E084DAB9 | apt-key add -
```

Note: I know this was required in September 2015 but may not be now.

To install R:

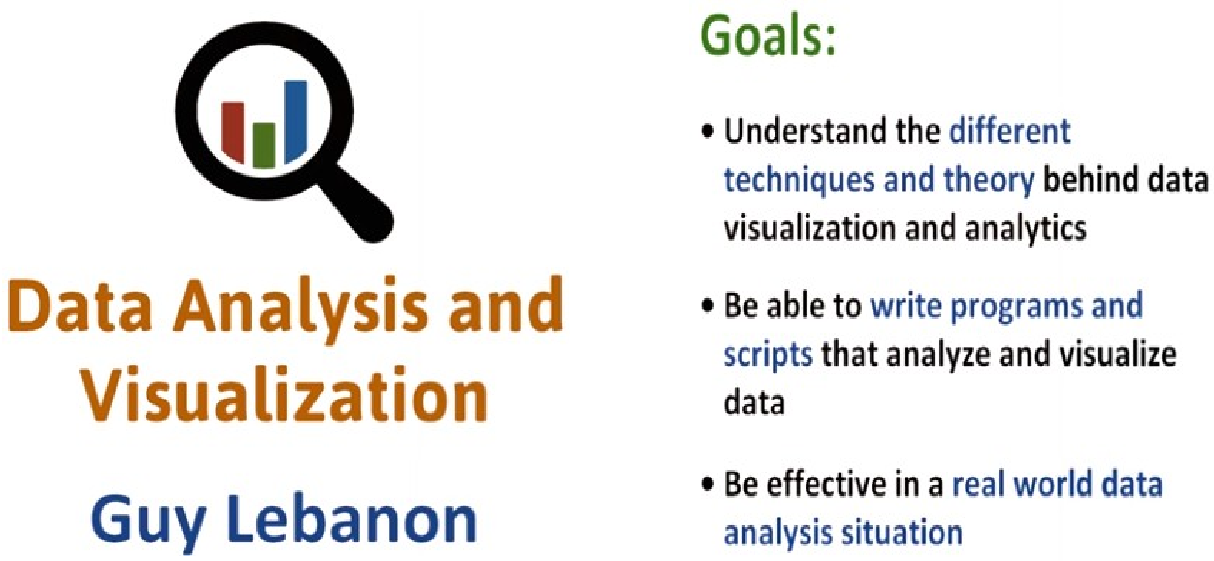
```
apt-get install r-base r-base-dev libatlas3-base littler r-cran-getopt
```

If you want to install python libraries:

```
apt-get install python-rpy python-rpy-doc python-rpy2
```

The student may find the following links useful for learning R:
Quick R
http://www.statmethods.net/
R Cookbook
http://www.cookbook-r.com/
R-Bloggers
https://www.r-bloggers.com/
StackOverflow About R
http://stackoverflow.com/tags/r/info
StackOverflow R FAQ
http://stackoverflow.com/questions/tagged/r-faq%20
Google's R Style Guide
(page not found)
Jupyter and Conda for R
(page not found)

There is an open R Basics lesson in the OMSCS Student Orientation.
Please feel free to do as much (or as little) of the lesson as you like.
R Basics Lesson in OMSCS Student Orientation
(Page not Found)



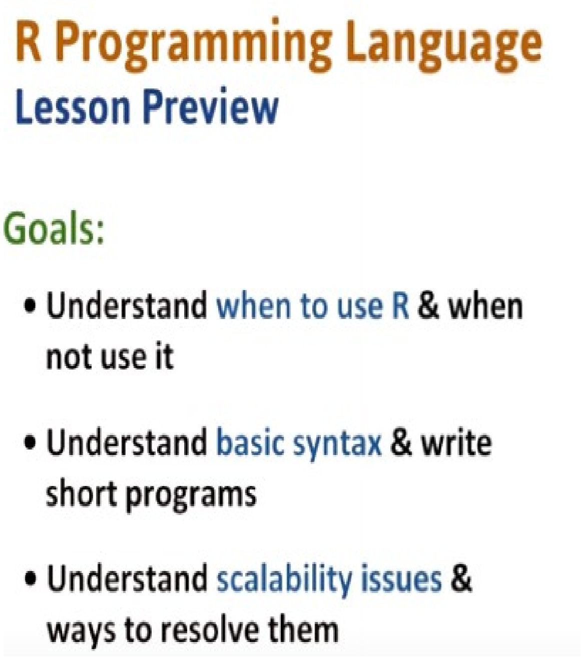
Data Analysis and Visualization

Guy Lebanon

Goals:

- Understand the different techniques and theory behind data visualization and analytics
- Be able to write programs and scripts that analyze and visualize data
- Be effective in a real world data analysis situation

1. Hello, I'm Guy Lebanon and this course is Data Analysis and Visualization. Data analysis and data visualization is a hugely important field. It is critical in search engines, social networks, e-commerce and many other areas in high tech. It is also very important to traditional industries like banking and insurance. I have been working in this field for close to 20 years in both academia and industry. And I'm happy to share my experience with you. In this course, we will achieve three goals. Understanding different techniques and theory behind data visualization and analytics. Be able to write programs and scripts that analyze and visualize data. And be effective in a real world data analysis situation. In this course, we will be using the R programming language. We will first learn how to use it, then learn how to visualize data and process it. Later on we will learn how to model data using logistic regression and linear regression. Our final lesson will be how to handle high dimensional data effectively using regularization. We'll do homework assignments to reinforce the ideas we're discussing. And we'll do a multi-stage project that I know you will find to be both challenging and interesting. This course will open the door to data, and allow you access to the powerful world of data analysis and visualization.

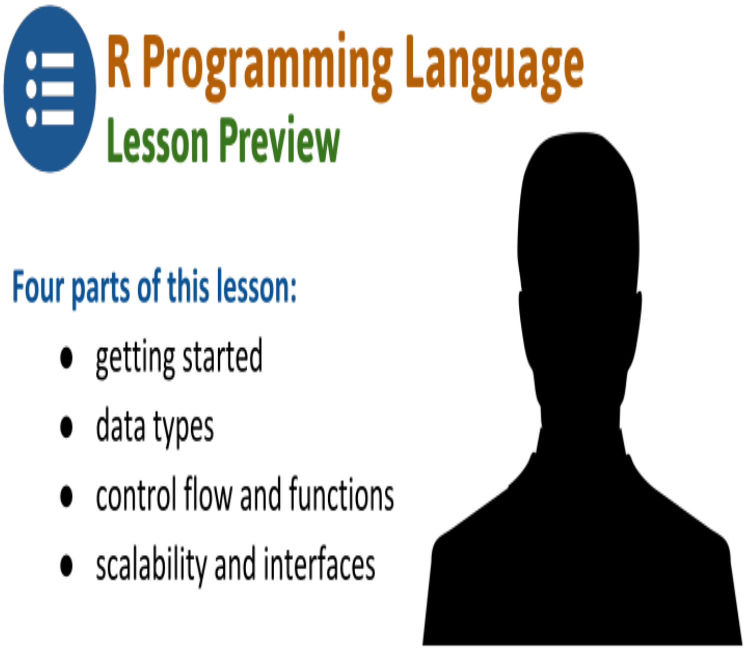


R Programming Language

Lesson Preview

Goals:

- Understand when to use R & when not use it
- Understand basic syntax & write short programs
- Understand scalability issues & ways to resolve them



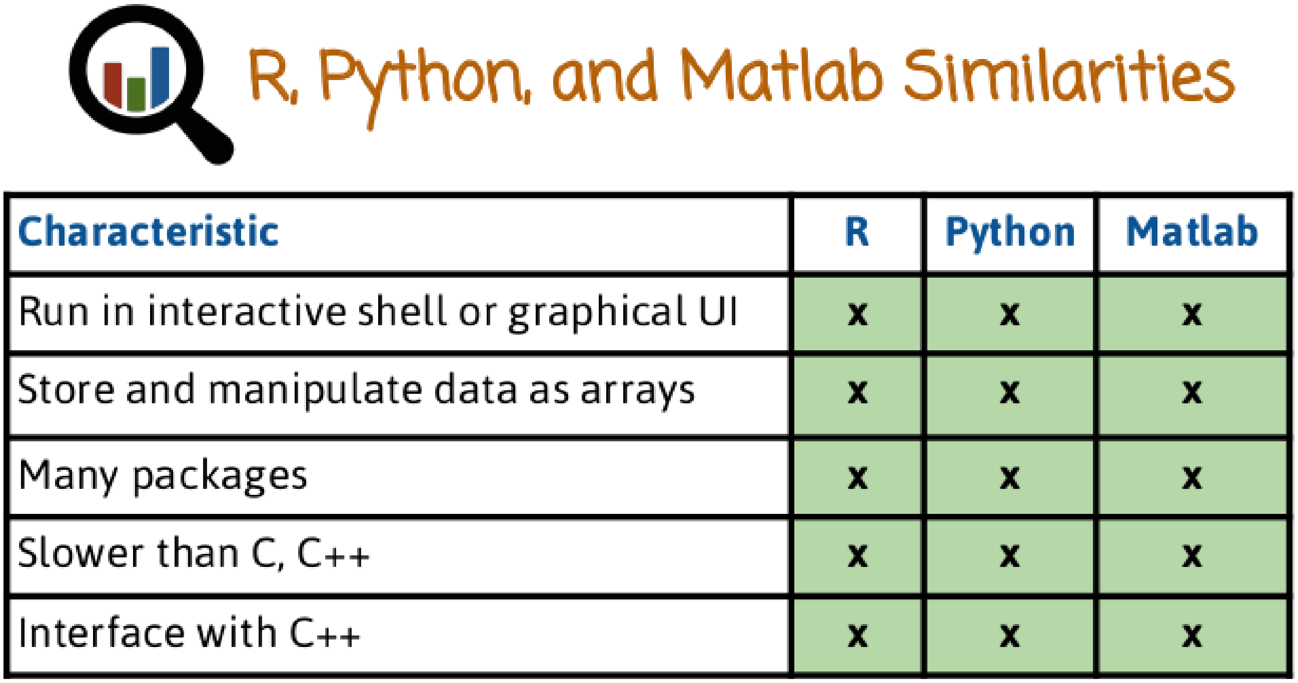
R Programming Language

Lesson Preview

Four parts of this lesson:

- getting started
- data types
- control flow and functions
- scalability and interfaces

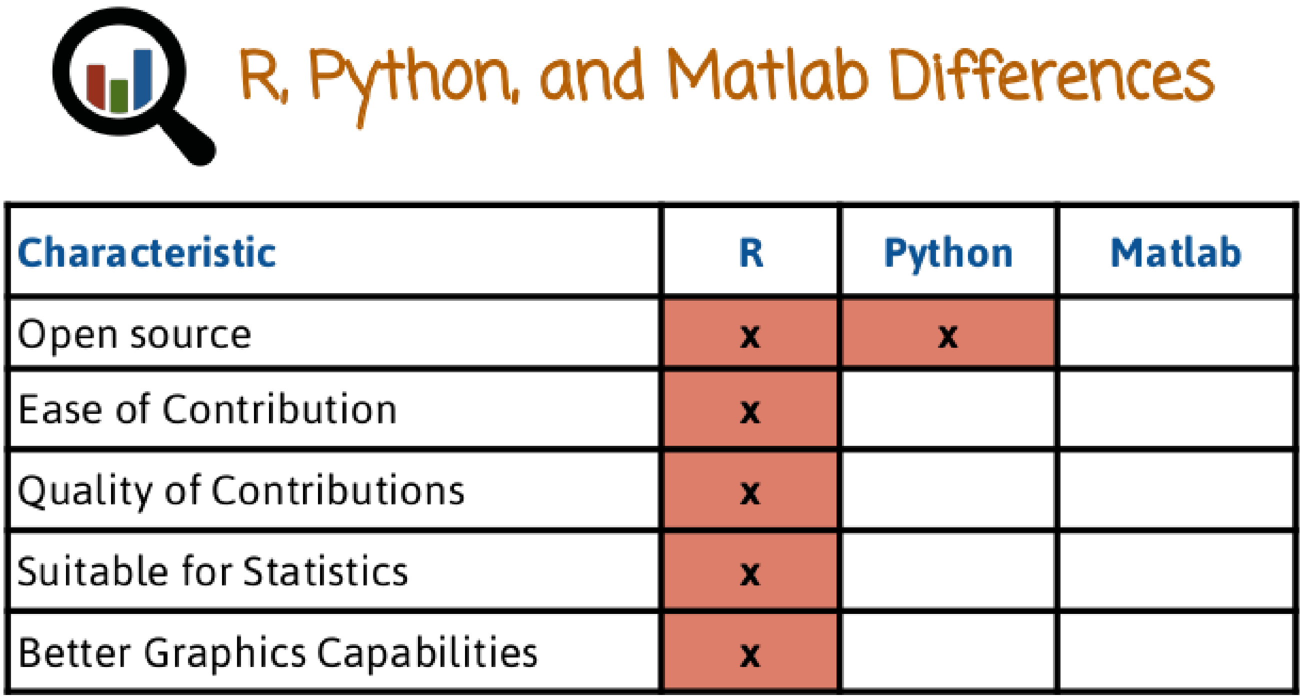
2. The R programming language is a useful tool for data scientists. Most production code in industry is written in languages such as C++ and Java that you may already know. Using these languages for data analysis and visualization is usually not very effective. R is much better suited in these cases as it is fast to develop code in it and experiment with it. But in many cases, engineers and scientists need a combination of R and a fast production language like C++ and Java. We will start with a brief description of R and how it is related to other languages, and then describe its basic syntax. As the lesson progresses, we will learn additional features of the language until at the end you will be able to write reasonably effective R programs. We will also spend some time going over scalability issues related to R and how to handle them. The lesson is divided into four sections. Getting started with R. Data types in R. Control flow and functions in R. And scalability and interfaces in R.



R, Python, and Matlab Similarities

Characteristic	R	Python	Matlab
Run in interactive shell or graphical UI	x	x	x
Store and manipulate data as arrays	x	x	x
Many packages	x	x	x
Slower than C, C++	x	x	x
Interface with C++	x	x	x

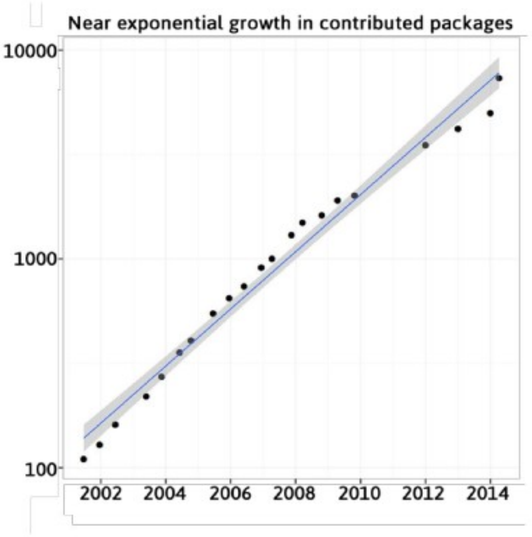
3. There are three popular languages for interactive data analysis, R, Matlab, and Python. Our decision of which language to use should be based on the task we're doing. All three languages run inside an interactive shell or a graphical user interface. They all emphasize storing and manipulating data as multidimensional arrays, which is very useful for data analysis. All three have many packages available that extend the original functionality of the language. All three languages are typically slower than C, C++, and Fortran, though there are several techniques that can help speed up things. All three languages can interface with native C++ code for speeding up bottle necks.

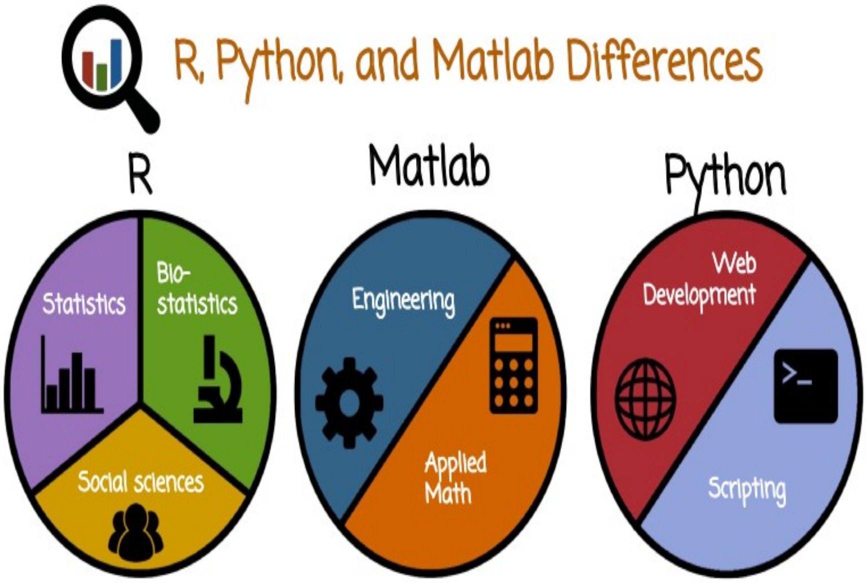


R, Python, and Matlab Differences

Characteristic	R	Python	Matlab
Open source	x	x	
Ease of Contribution	x		
Quality of Contributions	x		
Suitable for Statistics	x		
Better Graphics Capabilities	x		

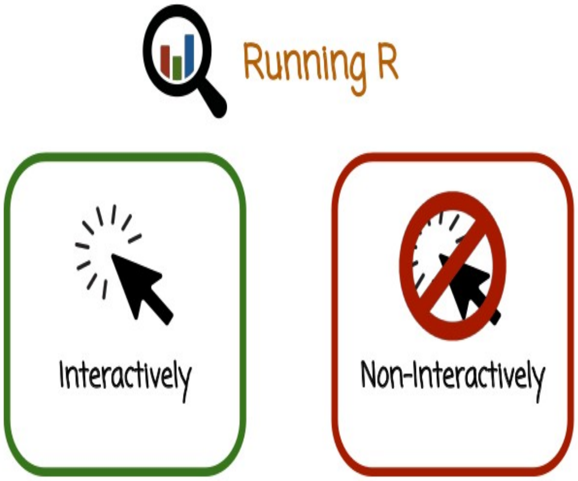
4. Now that we know how the three languages are similar, let's look at how they're different. R and Python are open source and free, Matlab is not. Contributing packages to R is a more standardized process.





R, Python, and Matlab Differences

R is seeing an incredible growth in the number of package contributions. In this graph, we see on the x-axis, the year. And in the y-axis, the number of contributed packages. The y-axis is displayed on the log scale and the linear trend here, shows an exponential increase in the number of contributed packages. R's standardized process of contributing packages has led to a large group of motivated contributors who contribute high quality packages. R syntax is more suitable for statistics and data, as it was originally designed for that purpose. Finally, R has better graphics capabilities than both Python and Matlab. R is popular in statistics, bio-statistics and social sciences communities. In particular, Matlab is very popular in engineering and applied math. Python is popular for data analysis. But in addition, it's also popular for web development and scripting.



Running R

Interactively

Non-Interactively

Running R - Interactively

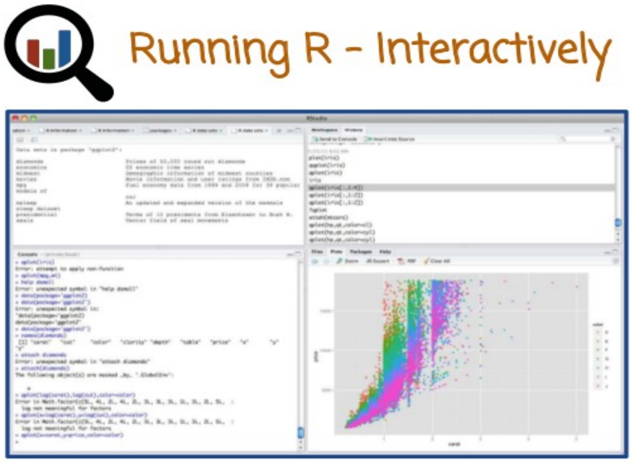
RStudio

5. There are two ways to use R, interactively and non-interactively. To run R interactively, we need to download and start the R application or the RStudio application. You can start the application by double-clicking the application icon. For example, in this case we see the RStudio application icon.

Running R - Interactively

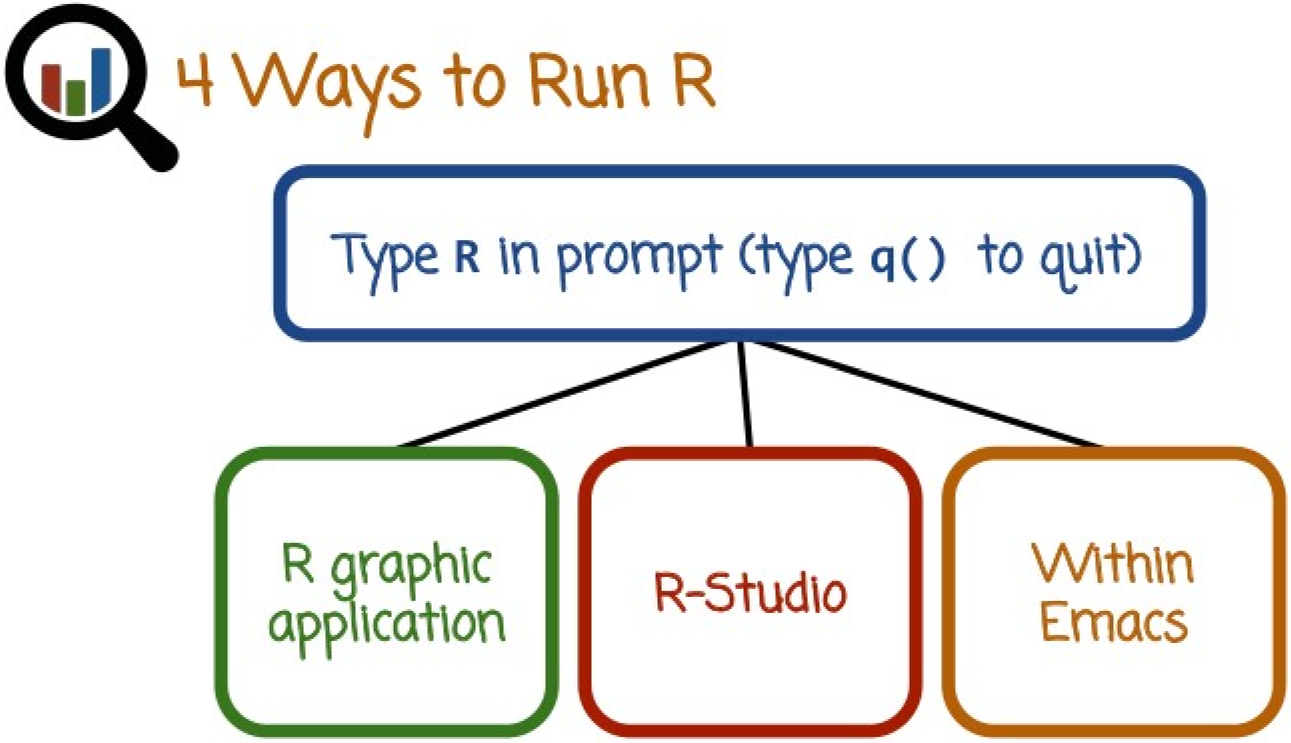
From the terminal:

MAC	$ open -a RStudio .
Linux	$ rstudio

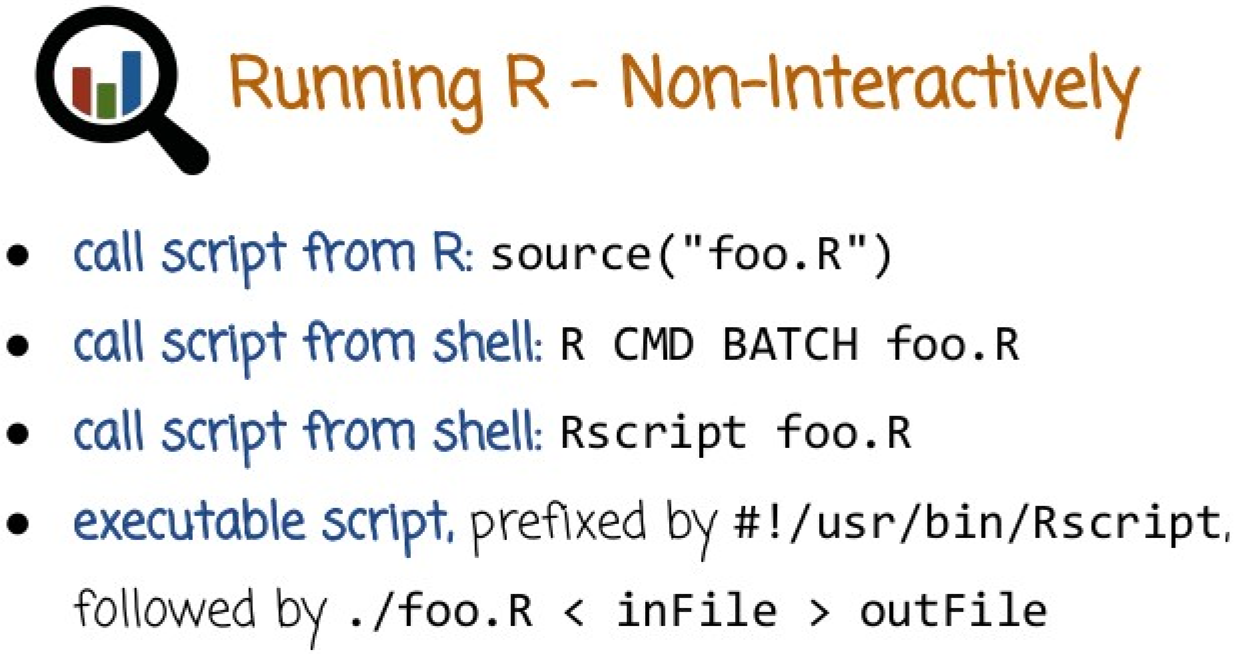


Running R - Interactively

You can also run R in the terminal window by typing R from the terminal program. You can start the RStudio graphical user interface by typing open -a RStudio in the terminal of MAC or typing rstudio in Linux's terminal. RStudio is a separate program that makes running R interactively particularly comfortable. Here is an example of what the RStudio application looks like. There are four panels that usually display the R Prompt, Plots and Help, History and Workspace, and the Editor window. The top left panel shows the Editor window and you can switch between multiple files by clicking on the tabs at the top. The top right panel shows either the workspace or the command history based on which tab you click on. The bottom left panel shows an interactive R command line window where you can type R commands and see their execution. The bottom right panel shows figures and help files. Again, depending on which of the tabs you click on.



Let's review again several ways of running R. The first way is to type R in the prompt in the terminal window. And to quit type q(). The second way to is use the R graphic application. The third way is to R-Studio graphic application. Notice that you need to download R-Studio separately after you download R. There are also other ways. For example, you can run R within Emacs using the Emacs Speaks Statistics package. You can use whichever of these methods you prefer. Many beginners particularly find R-Studio the most convenient.



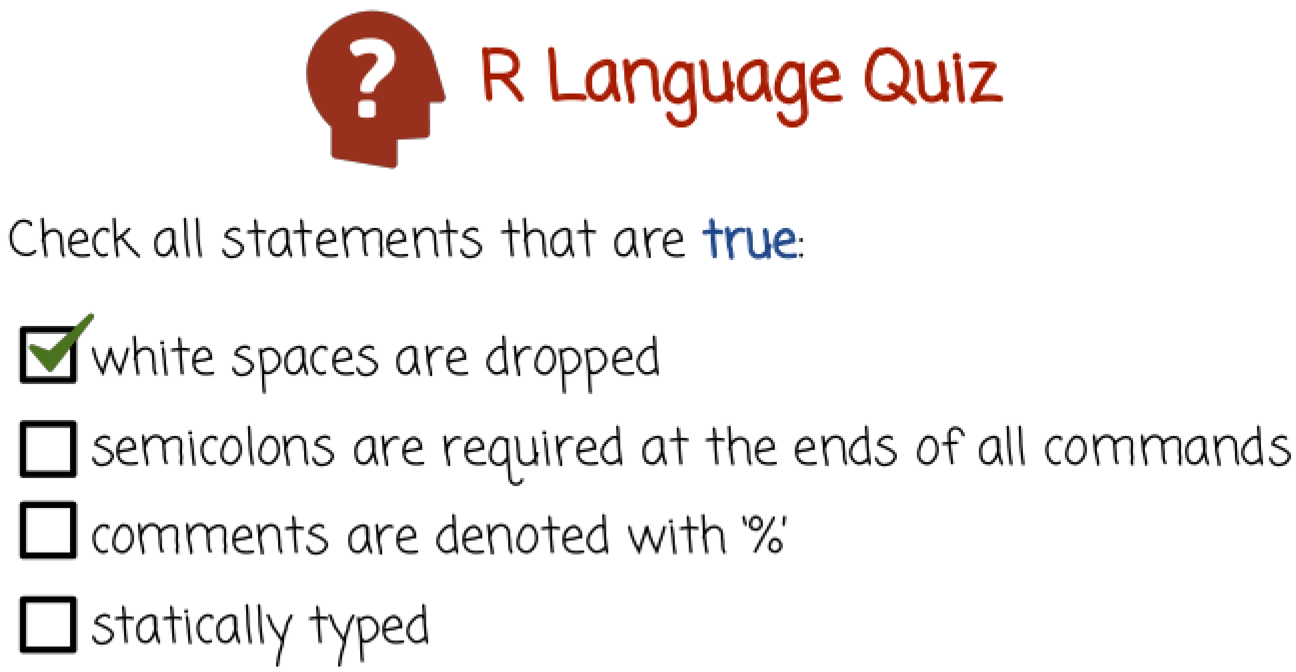
Running R - Non-Interactively

- call script from R: `source("foo.R")`
- call script from shell: `R CMD BATCH foo.R`
- call script from shell: `Rscript foo.R`
- executable script, prefixed by `#!/usr/bin/Rscript`, followed by `./foo.R < inFile > outFile`

在我電腦上, 第一種方法不行, 第二和第三種方法可以, 第四種沒試

We can also run R non-interactively. To do that we need to create a file of R instructions called the script file. The convention is that file name should end with a .R file name extension. After we create the file, we can execute it by typing the command source with an argument, the name of the file from within R. Then R will execute the instructions in the file, line by line. We can also run R non-interactively from the terminal window, without opening first the R program. Here is an example of doing that. We type at the terminal window R CMD BATCH then the name of the script file. Another way of doing the same thing is to use the Rscript command, again from the terminal window, without opening R first. Finally, we can create an executable script file containing our commands that is prefixed by this line right here. And then we can execute the file from the command prompt without starting R first. By just typing the name of the executable file prefixed by its directory. This option is the most flexible one since you can use input and output redirections for example here. But each of this four options is suitable and depends on whichever one you find the most convenient.

6. This is our first quiz on the R language. Check all statements that are true. White spaces are dropped. Semicolons are required at the ends of all commands. Comments are denoted with %. R is statically typed.



R Language Quiz

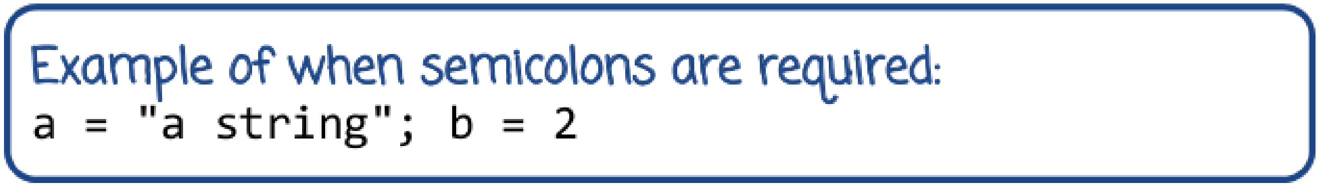
Check all statements that are **true**:

☑ white spaces are dropped

□ semicolons are required at the ends of all commands

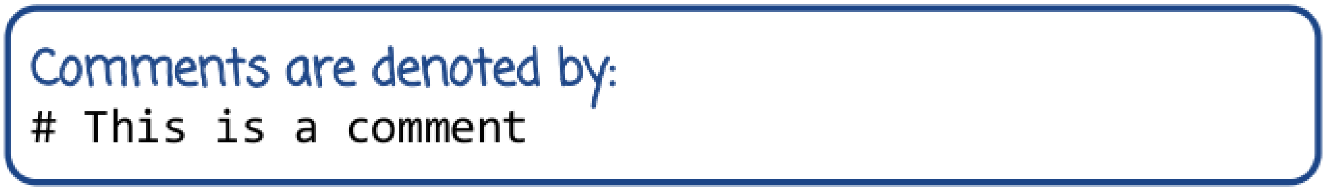
□ comments are denoted with '%'

□ statically typed



Example of when semicolons are required:

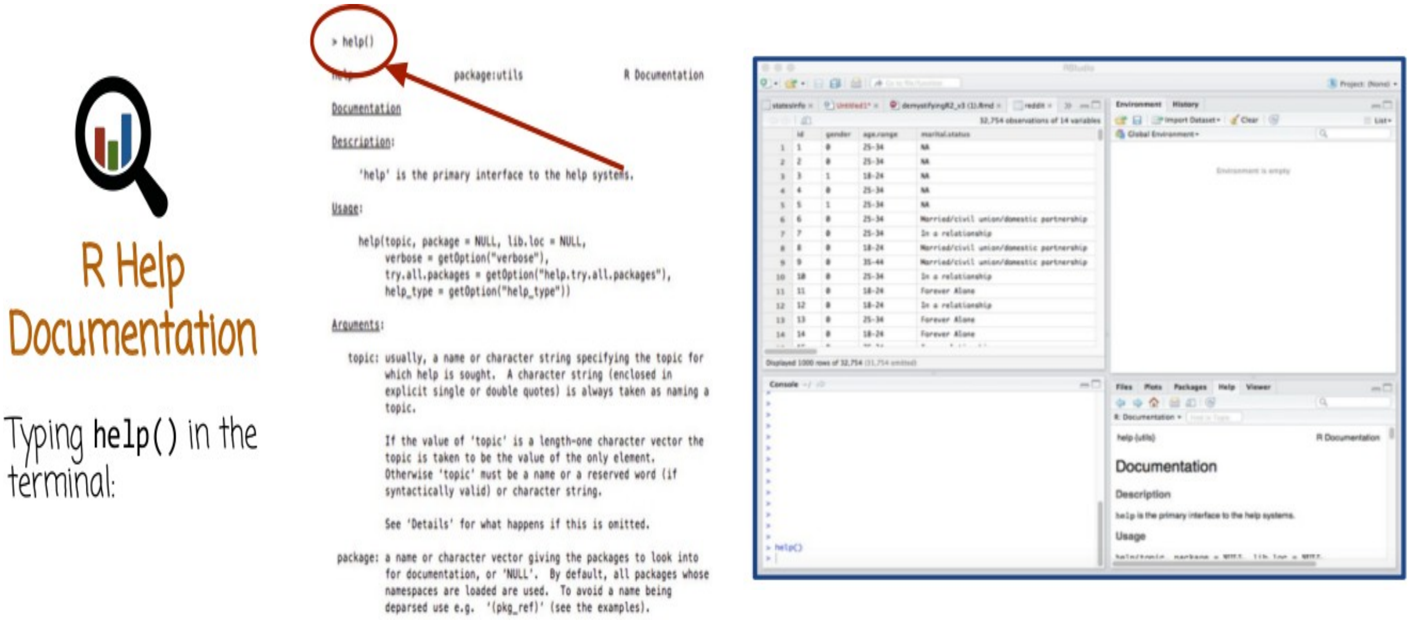
```
a = "a string"; b = 2
```



Comments are denoted by:

```
# This is a comment
```

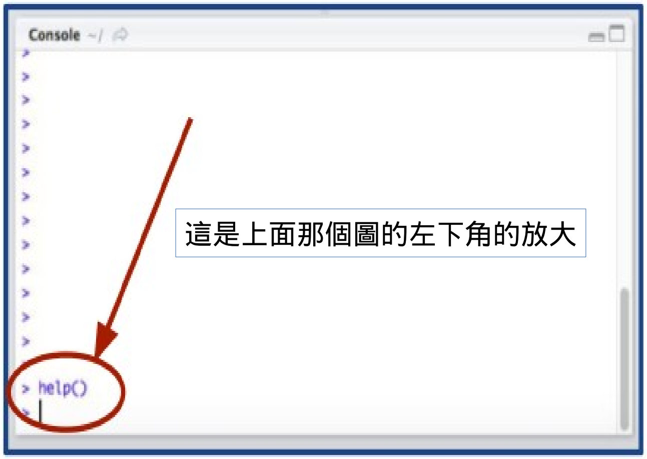
7. The first statement is true, white spaces are dropped. R does not use indentation like python. Semicolons are not required at the end of all commands. They are required if you want to put two commands in the same line, then you need a semicolon in between the two commands. But notice we may or may not put a semicolon at the end of the last command. Comments are not denoted with the percent sign, they're denoted with the #. Statically typed 原來是這個意思 → Finally R is not statically typed, meaning that the type of the variable can be modified in runtime, rather than being determined during compilation time.



R Help Documentation

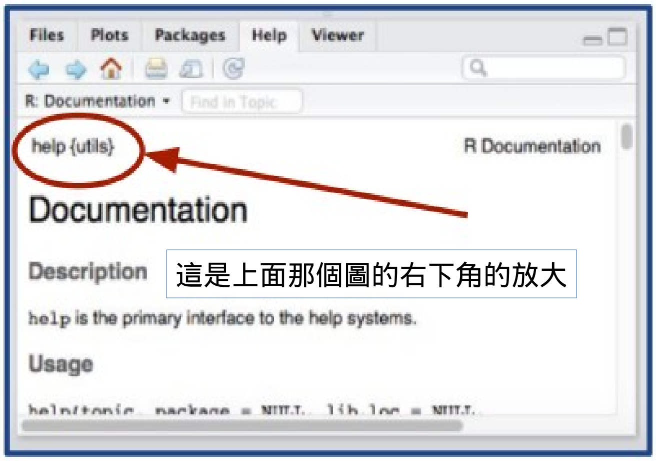
Typing help() in the terminal:

```
> help()

help                    package:utils                    R Documentation

Documentation

Description:

     'help' is the primary interface to the help systems.

Usage:

     help(topic, package = NULL, lib.loc = NULL,
          verbose = getOption("verbose"),
          try.all.packages = getOption("help.try.all.packages"),
          help_type = getOption("help_type"))

Arguments:

   topic: usually, a name or character string specifying the topic for
          which help is sought.  A character string (enclosed in
          explicit single or double quotes) is always taken as naming a
          topic.

          If the value of 'topic' is a length-one character vector the
          topic is taken to be the value of the only element.
          Otherwise 'topic' must be a name or a reserved word (if
          syntactically valid) or character string.

          See 'Details' for what happens if this is omitted.

 package: a name or character vector giving the packages to look into
          for documentation, or 'NULL'.  By default, all packages whose
          namespaces are loaded are used.  To avoid a name being
          deparsed use e.g.  '(pkg_ref)' (see the examples).
```



```
> help()
```

這是上面那個圖的左下角的放大



help {utils}　R Documentation

Documentation

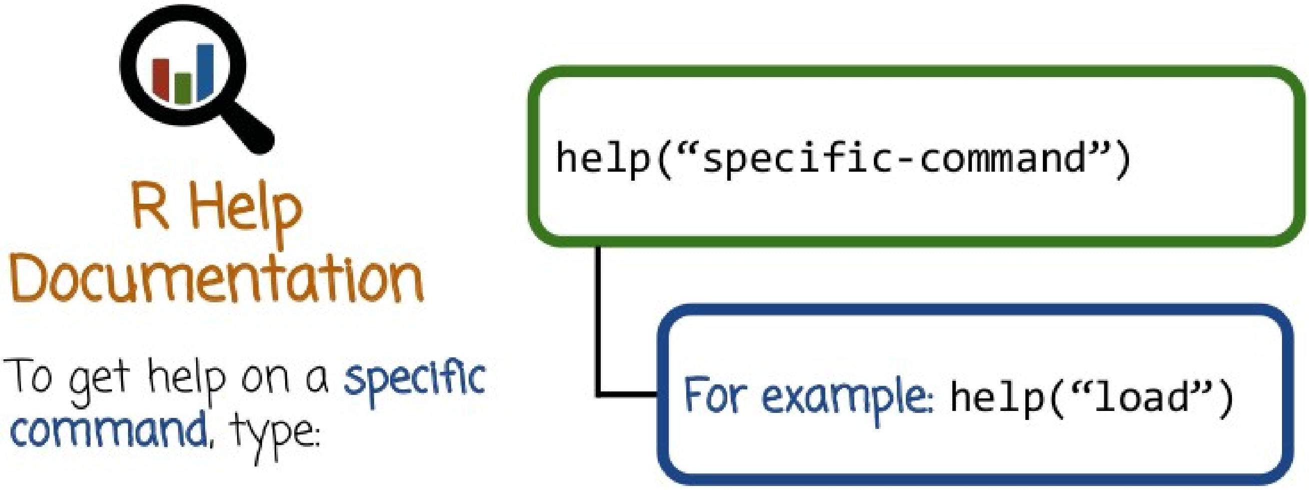
Description

help is the primary interface to the help systems.

Usage

```
help(topic, package = NULL, lib.loc = NULL,
```

這是上面那個圖的右下角的放大



R Help Documentation

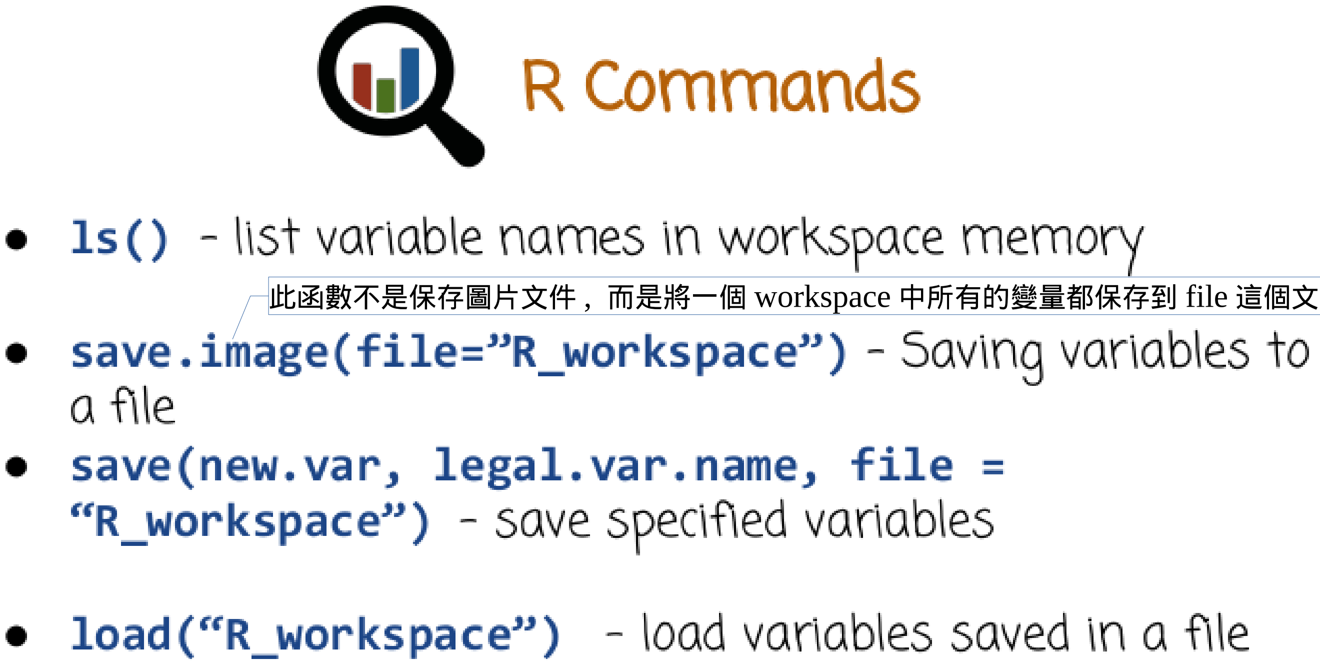
To get help on a specific command, type:

```
help("specific-command")
```

For example: help("load")

8. R has extensive documentation that you can access using the Help function. When you use the Help

function, you can pass it an argument corresponding to the name of a function or the name of a dataset, and R will display the corresponding documentation. This is what it looks like when you type help after starting R in the terminal. If you type help in R Studio, notice you type help in the bottom left panel and the help window on the bottom right will show the output. Here is zooming in on the bottom left panel, typing help, and here is zooming in on the bottom right panel, where we see the documentation appearing. Here is an example of getting help on a specific function, in this case the function load.

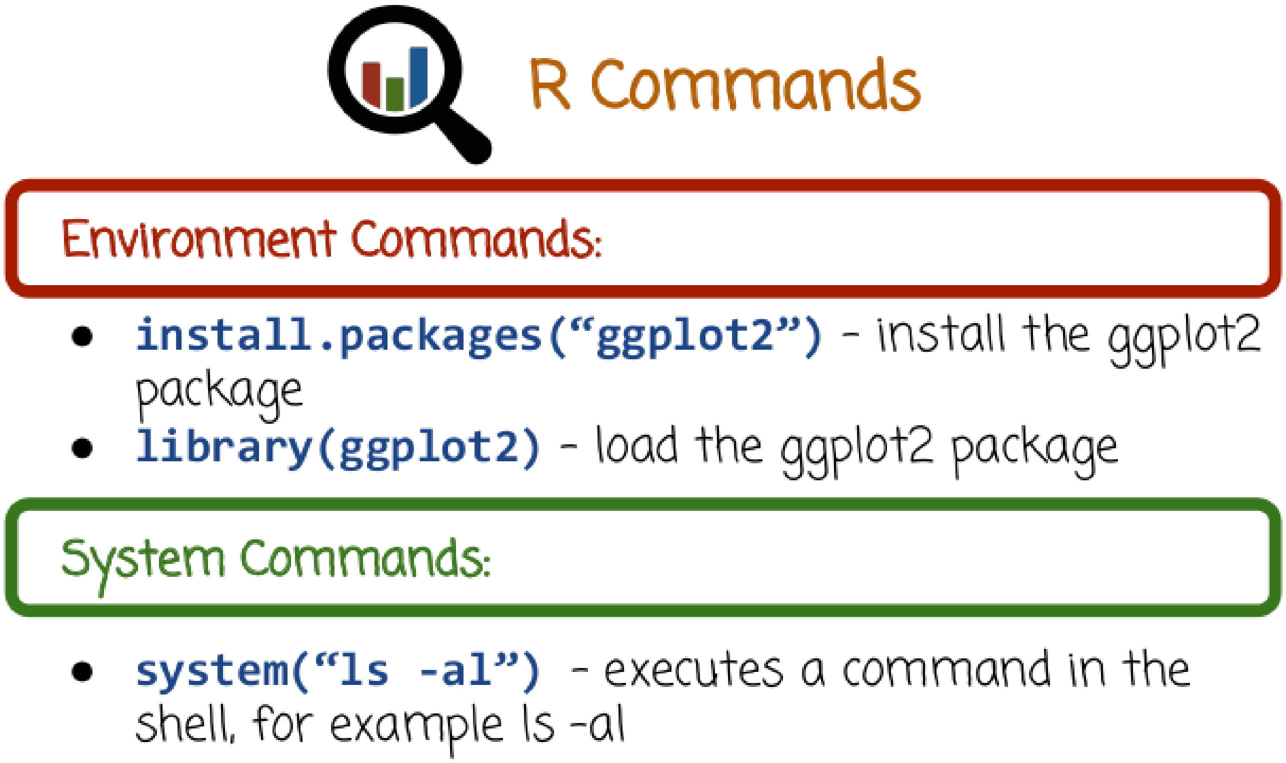


R Commands

- ls() - list variable names in workspace memory
- save.image(file=”R_workspace”) - Saving variables to a file
- save(new.var, legal.var.name, file = “R_workspace”) - save specified variables
- load(“R_workspace”) - load variables saved in a file

此函數不是保存圖片文件，而是將一個 workspace 中所有的變量都保存到 file 這個文件中 .

9. Here are some very useful commands. Ls in R lists variable names in the workspace memory. Notice that in R, we need to have open parentheses closed parentheses after ls and other similar instructions. The reason for that is that R is a functional language in which statements such as ls and others are functions rather than non-function literals. The function save.image saves (all) variables in a (current) workspace to a file. In this case, we need to pass the file name (即 R_workspace) as an argument, which is assigned to the argument named “file”. Notice how the period here is a part of the function name rather than an operator as it is in other languages like C++ and Java (變量名也可以用點, 比如上圖中的 new.var). In other words, we can have a variable name or a function name separated by one or more period to enhance availability. If we only want to save a few variables rather than the entire workspace, we can use the function save. Inside the function argument list we need to specify the variables that we want saved together with the file name. To reverse operation of loading variables from a file into the workspace is done by the function load. Notice how in R when you pass arguments to a function, you can pass the argument names. In this case, new.var and legal.var.name are just variables that you want to pass to the function. The string R_workspace is another variable that you want to pass to the function, but we have here file = R_workspace, which instructs R to bind the string R_workspace to the argument inside a function called file. This kind of binding allows us to pass arguments out of order. If we keep the right order, we do not need to specify the name of the argument your file. And that's why, in the last example, we just passed the name of the file, R_workspace without typing file equals R_workspace.



R Commands

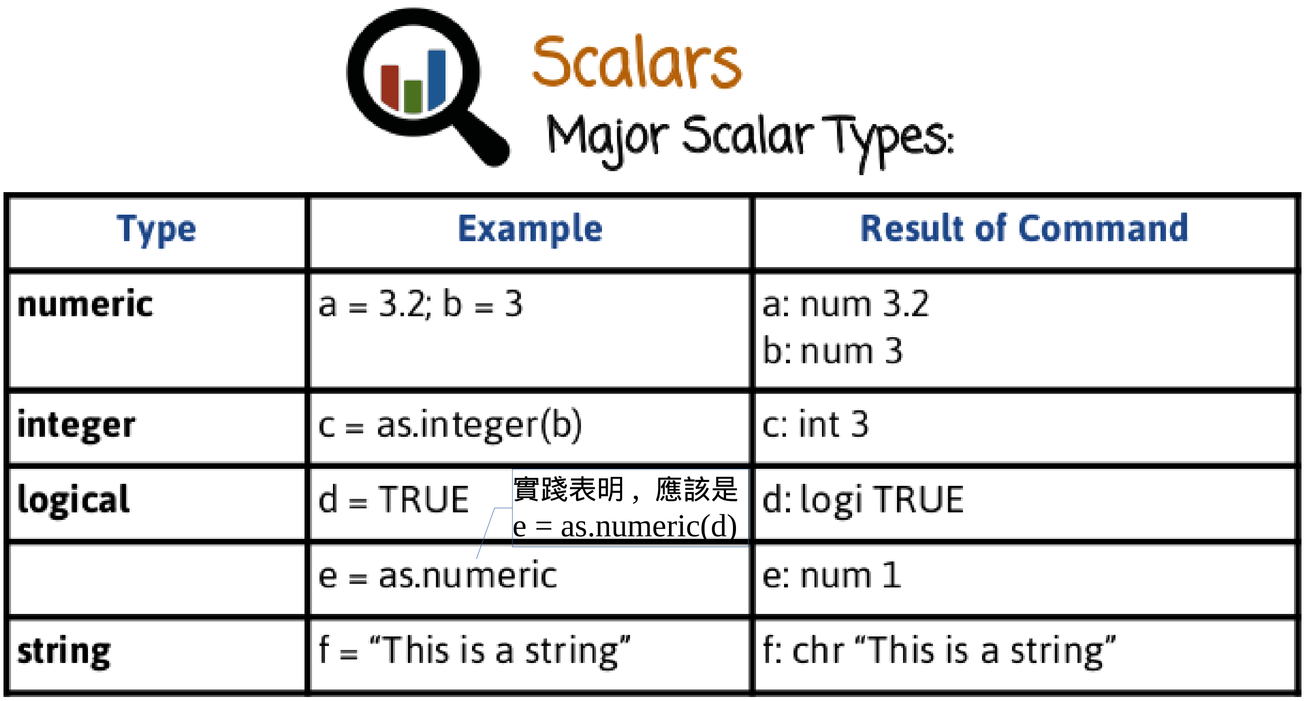
Environment Commands:

- `install.packages("ggplot2")` - install the ggplot2 package
- `library(ggplot2)` - load the ggplot2 package

System Commands:

- `system("ls -al")` - executes a command in the shell, for example ls -al

We saw before that R has many packages. To access them, you first need to install the package using the install.packages function where we pass the package name, as a parameter to the function. After we install the package, in this case ggplot2 package, we need to bring it into scope using the function library. If we install the package but do not bring it into scope, I will not be able to recognize the functions and datasets in the package. We can use the system function in R to execute a command in the underlying shell. For example, if we want to execute the command ls-al, which lists all files In the current directory and we want to see the output of that command inside the R window, we can type the command right here, system("ls -al"). The output of the shell command will be displayed in the R window.



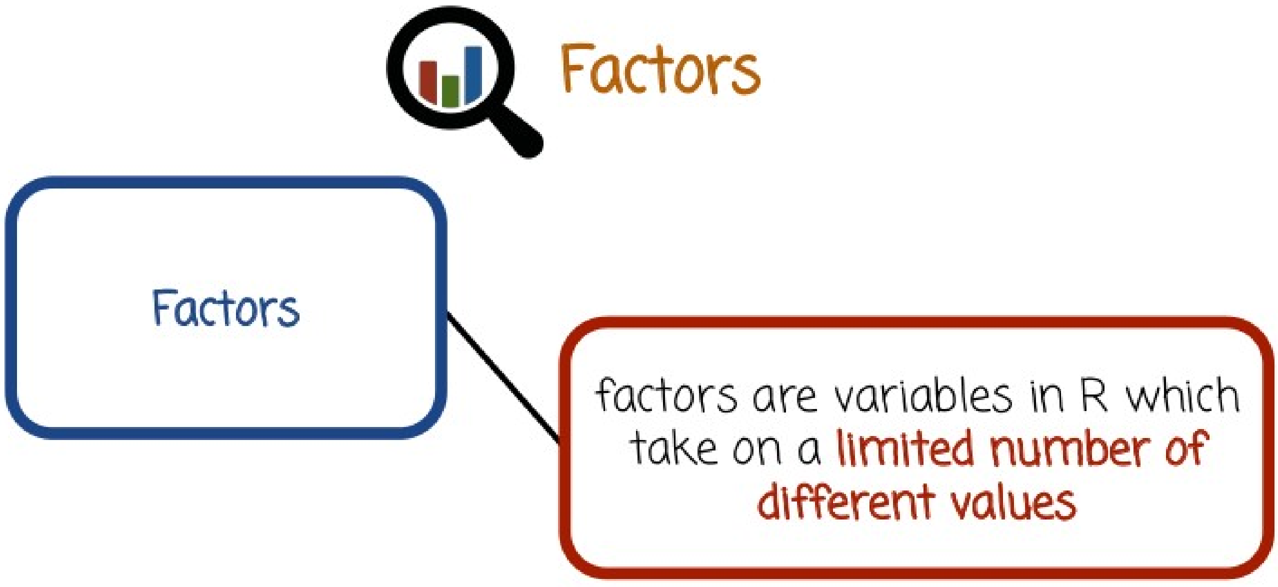
Scalars

Major Scalar Types:

Type	Example	Result of Command
numeric	a = 3.2; b = 3	a: num 3.2 b: num 3
integer	c = as.integer(b)	c: int 3
logical	d = TRUE	d: logi TRUE
	e = as.numeric	e: num 1
string	f = "This is a string"	f: chr "This is a string"

實踐表明，應該是
e = as.numeric(d)

10. Numeric data types are the most common scalar type. They correspond to floating point values. Here's an example. The variable a is assigned the value 3.2, and the variable b is assigned the value 3. Integers are somewhat less common in R. Here's an example of defining an integer. We take a numeric variable and we cast it using the as.integer function to a variable c. Logical type corresponds to true or false. Here's an example where we assign the value true to a variable d. In this case, we take the logical type in the variable d and we cast it to a numeric variable e. In this case, we casted true into the numeric value 1. Here is an example of a string data type. We can use either double quotation or single quotation in R.



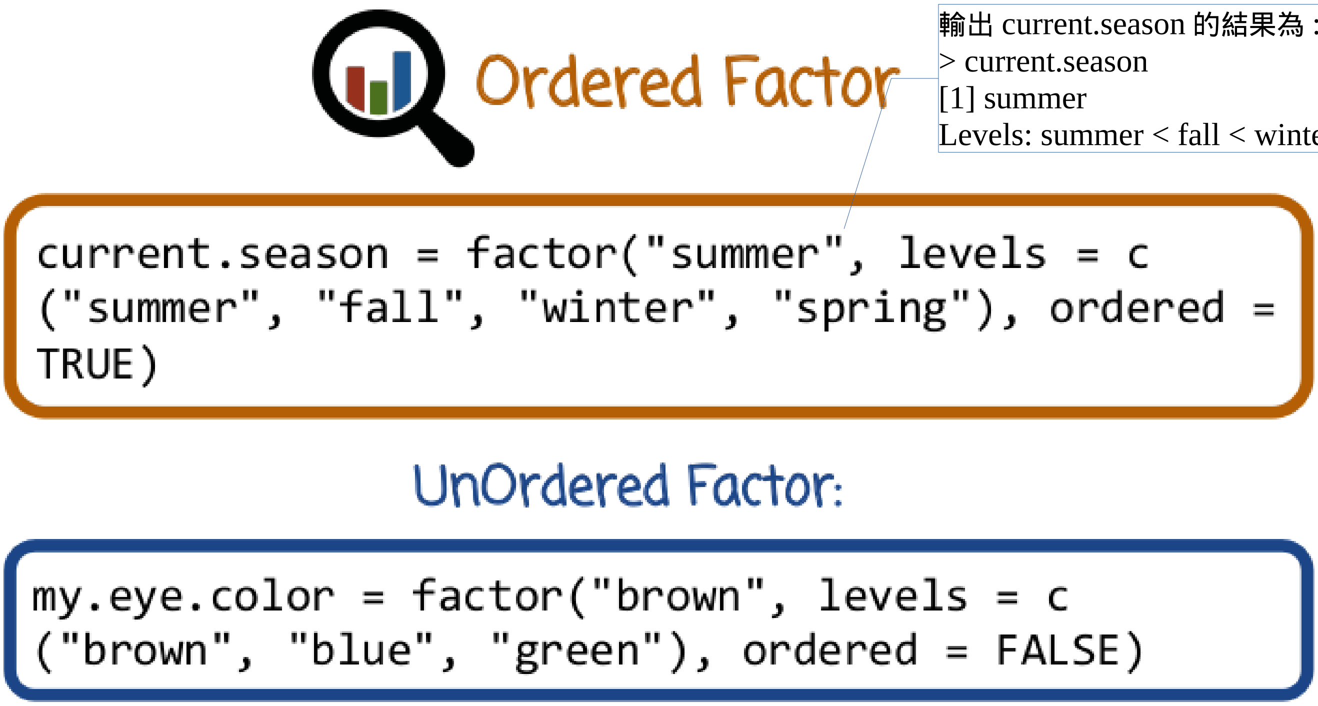
Factors

Factors

factors are variables in R which take on a limited number of different values

11. Factors are variable in R which take on a limited number of different values. They're similar to

enums in C++ and Java.



Ordered Factor

```
current.season = factor("summer", levels = c
("summer", "fall", "winter", "spring"), ordered =
TRUE)
```

UnOrdered Factor:

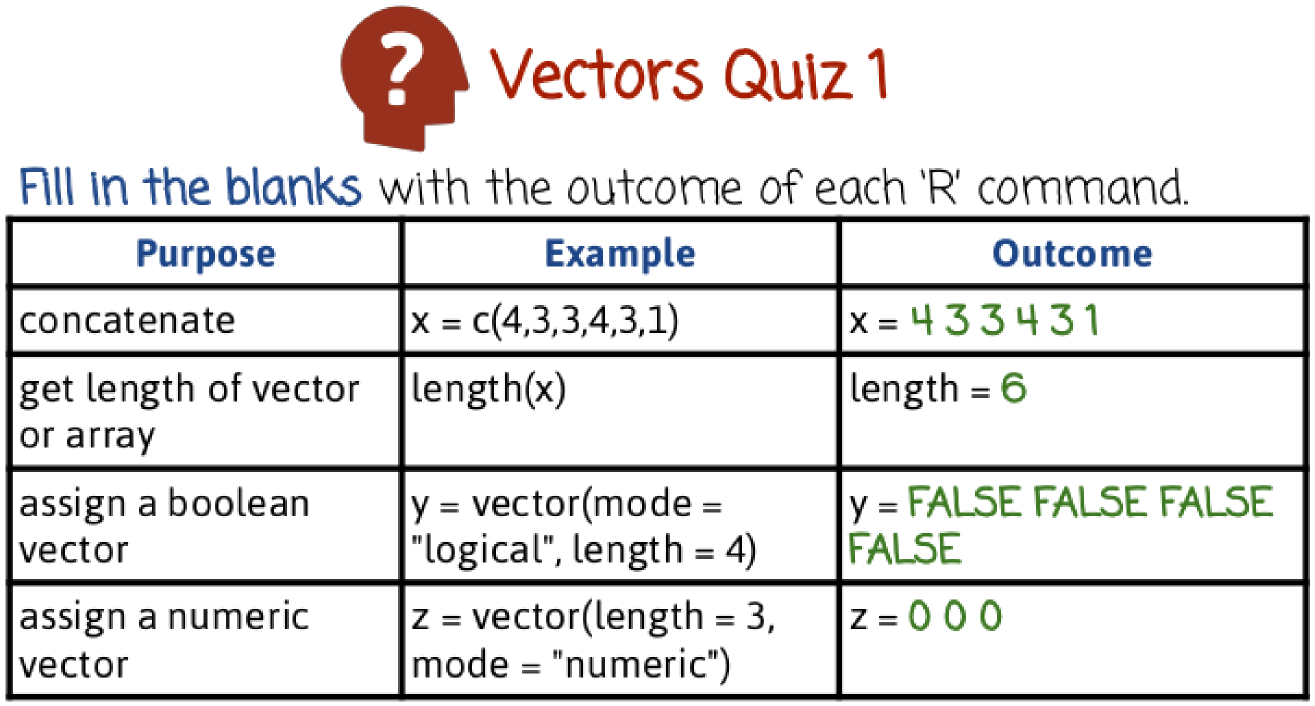
```
my.eye.color = factor("brown", levels = c
("brown", "blue", "green"), ordered = FALSE)
```

輸出 current.season 的結果為 :

```
> current.season
[1] summer
Levels: summer < fall < winter < spring
```

Factors can be ordered, in which case we need to define the ordering. Here is an example. We defined a variable called current.season and we want to make sure R knows it's a factor. So we call the factor function, we pass as the first argument the value of the factor variable, which is summer. As a second parameter, we pass all possible levels or values that the factor can take. In this case, it's a vector concatenating the string ("summer", "fall", "winter", "spring") (後面說了: c()函數就是 concatenate 的作用). The last argument indicates that this factor is ordered, by assigning the value TRUE to the parameter ordered. In this case the ordering is the default ordering in the concatenated vector of strings right here. Factors can also be unordered. Here is an example. We defined the variable called my.eye.color, and we assign it the value "brown", where the levels of the factor, or the possible values are brown, blue, and green. In this case, we indicate that the factor is not ordered.

12. Here is a quiz about vectors. This table has four rows. In each row we have an example command that attempts to achieve a specific purpose. What you need to do is fill in the outcome.



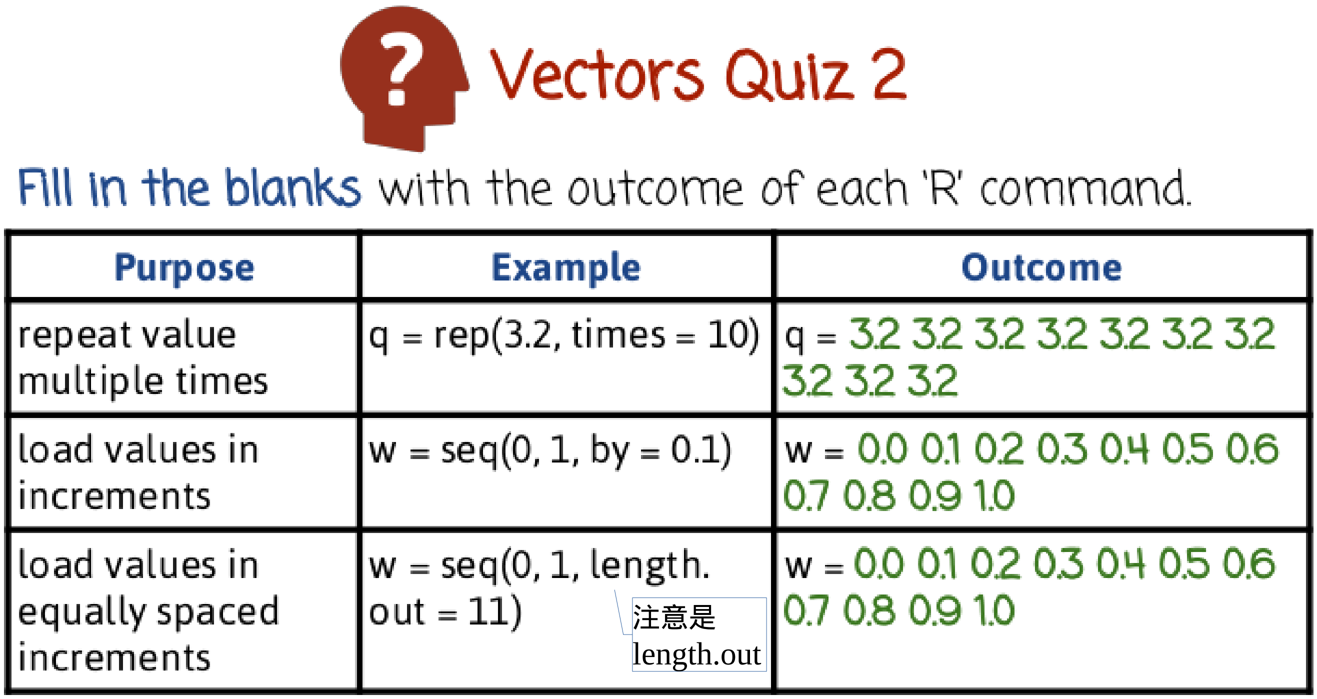
Vectors Quiz 1

Fill in the blanks with the outcome of each 'R' command.

Purpose	Example	Outcome
concatenate	x = c(4,3,3,4,3,1)	x = 4 3 3 4 3 1
get length of vector or array	length(x)	length = 6
assign a boolean vector	y = vector(mode = "logical", length = 4)	y = FALSE FALSE FALSE FALSE
assign a numeric vector	z = vector(length = 3, mode = "numeric")	z = 0 0 0

13. In the first row, we have the c function which stands for concatenate. What it does, it takes its argument, in this case 4,3,3,4,3,1 and creates a vector of values 4,3,3,4,3,1 and assign it to the variable x. In the second row, we wish to get the length of a vector or an array using the length function. Since the vector has six values, the outcome of the command would be six. The third row shows an example where we create the vector of logical variables of length 4. We see here that the vector is assigned false default value to all the logical variables in it. And that is because false is the default value of the logical type. In the last example, we create a numeric vector of length 3. The outcome is 0, 0, 0 again, because the default value of the numeric type is 0.

14. In this quiz, we have three rows, where in each row we execute the following code attempting to achieve a specific purpose. Please fill in the right column with the outcome of the commands.



Vectors Quiz 2

Fill in the blanks with the outcome of each 'R' command.

Purpose	Example	Outcome
repeat value multiple times	q = rep(3.2, times = 10)	q = 3.2 3.2 3.2 3.2 3.2 3.2 3.2 3.2 3.2 3.2
load values in increments	w = seq(0, 1, by = 0.1)	w = 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
load values in equally spaced increments	w = seq(0, 1, length. out = 11)	w = 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

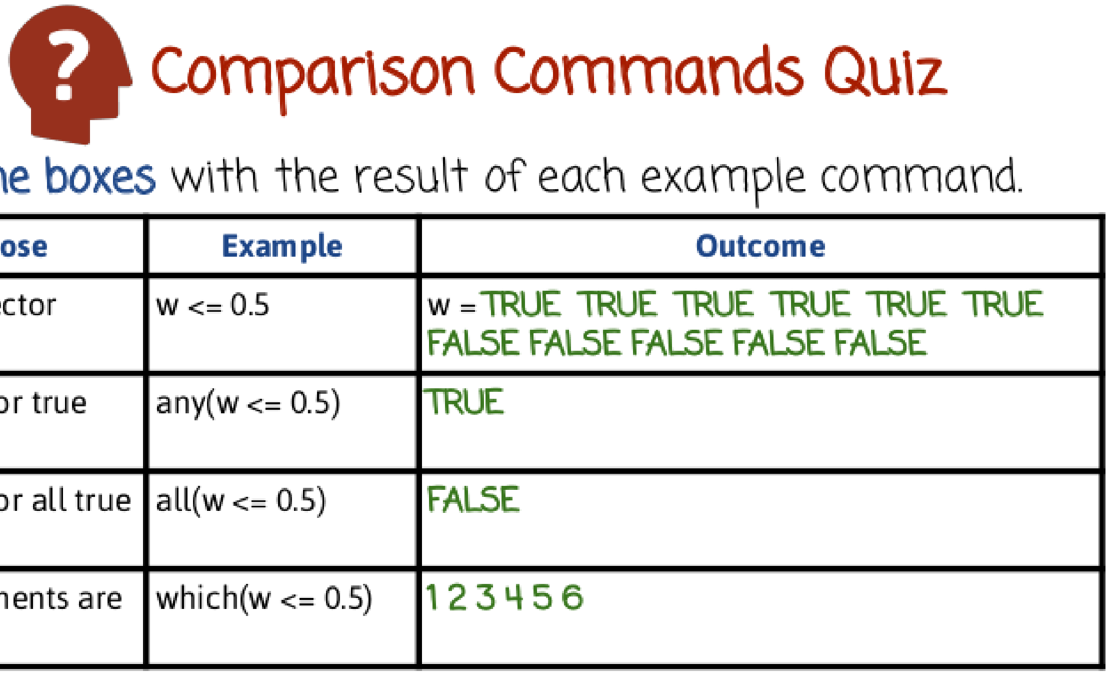
注意是 length.out

注意 R 中 seq(0, 1, ...)產生的結果中包含 0 和 1. 這跟 Java 中的 substring()和 Python 中的 range()都不同.

15. In the first row, we try to create a vector that has multiple repeated values. In this case we want to have the value 3.2 repeated ten times. The first argument being the value we want to repeat and the second argument being how many times we want to repeat it. In this case we assign it to the argument times so the function knows how to interpret it. We see here that the outcome is a vector of ten values 3.2. In the second row, we try to create a vector that spans numbers between zero and one, with increments of 0.1 in between consecutive numbers. The outcome is a vector of value 0.0, 0.1, 0.2, and so on until 1.0. The third line achieves the same goal as the second line, but instead of calling the function seq with an increment argument, we call the function with an argument specifying how many values we want the vector to contain. In this case we basically say we want 11 values between zero and one that are equally spaced and we get the same vector. Creating such sequences of values that span to extreme values is very important in order to create grids that can be used to graph data and functions.

16. In this quiz we have four rows. Where in each row we execute specific code trying to accomplish a specific purpose. Please fill in the right column with the outcome of the code. Please note, that the vector w is a vector of values between 0 and 1 with increments of 0.1, the same vector we saw in the previous quiz, specifically a vector containing the values, 0, 0.1, 0.2, etc until 1.

以下的所有 quiz 中, 都有 w = 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0:



Comparison Commands Quiz

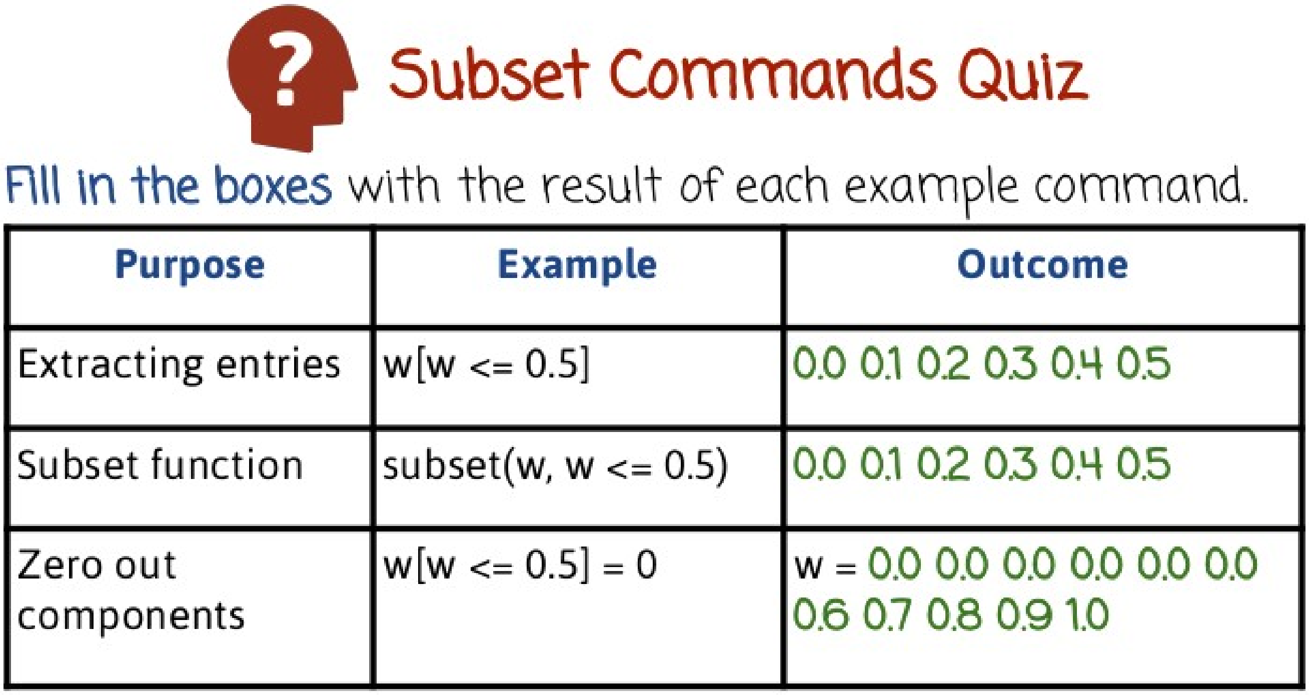
Fill in the boxes with the result of each example command.

Purpose	Example	Outcome
Boolean vector	w <= 0.5	w = TRUE TRUE TRUE TRUE TRUE TRUE FALSE FALSE FALSE FALSE FALSE
Checking for true elements	any(w <= 0.5)	TRUE
Checking for all true elements	all(w <= 0.5)	FALSE
Which elements are true	which(w <= 0.5)	1 2 3 4 5 6

注意 R 中的 index 是從 1 開始的, 而不是從 0 開始!
# Python uses 0-based indexing
# R uses 1-based indexing

17. In the first case, we have w <= 0.5. What happens, is that R compares the logical test whether the value is <=0.5 to each one of the values in the vector. Now remember the vector is 0, 0.1, 0.2, 0.3, all the way to 1. And when you run this logical test on each value of the function, you get at the beginning values are all true, where the vector elements are smaller than or equal to 0.5 and later on they become false as the values get larger. The second row, takes the result of the first row which is a vector of true and false values and calls the function any. The function any returns true, if any of the logical values in the vector is true, returns true as well. In this case, some of the elements of the value w <= 0.5 is we see in this line are true, therefore the function any will return true as well. In the third row, we check whether all the values and the logical vector are false. Since the vector w less than or equal to zero point five is partially true and partially false, the outcome of this command will be false. The last row, extracts the indices of the logical vector in which the logical value is true. Remember again, the value of the vector w <= 0.5 is written here. The first six values are true and the last five values are false. Therefore, the function which will return indices 123456 which are the indices of the vector w <= 0.5 where we have the value true.

18. Please fill out the right column with the outcome of running the code in the middle column. Remember, again, that the vector w is a vector of values 0.0, 0.1, 0.2, 0.3, and so on until 1. The same vector we used in the previous quiz.

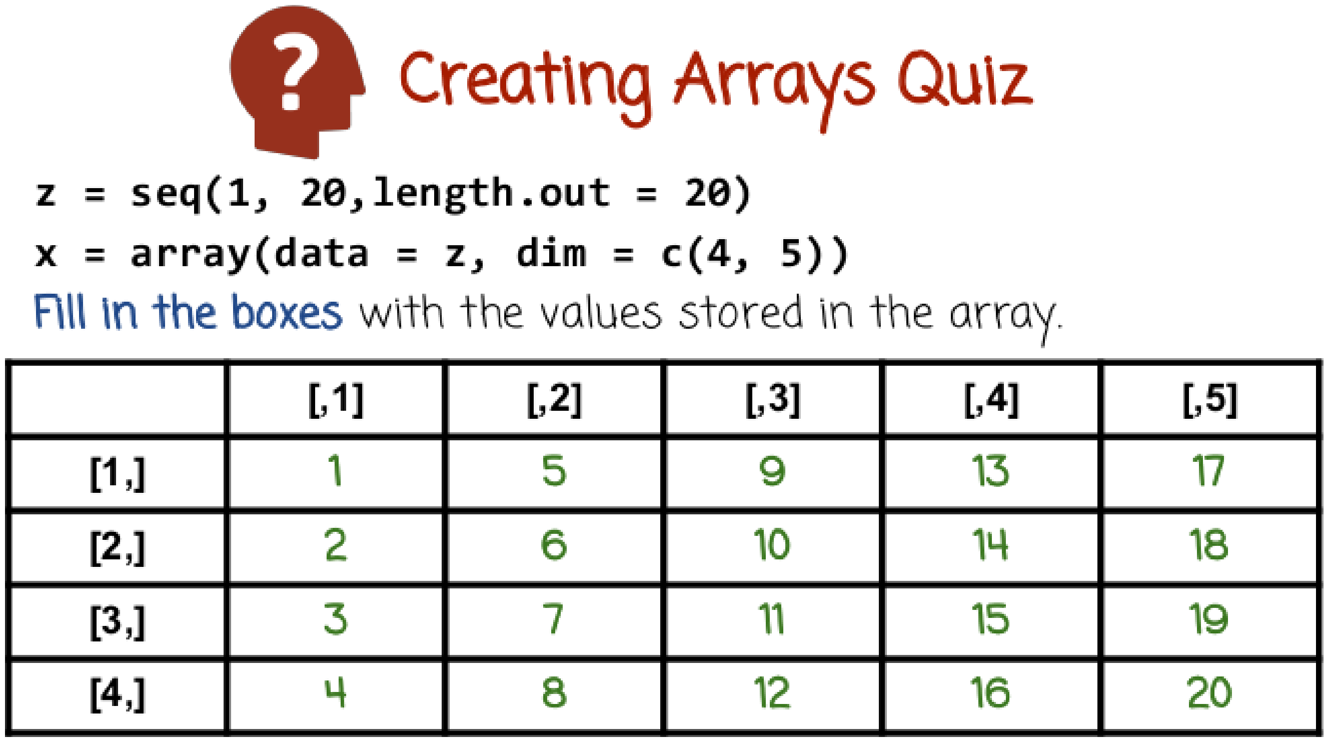


Subset Commands Quiz

Fill in the boxes with the result of each example command.

Purpose	Example	Outcome
Extracting entries	w[w <= 0.5]	0.0 0.1 0.2 0.3 0.4 0.5
Subset function	subset(w, w <= 0.5)	0.0 0.1 0.2 0.3 0.4 0.5
Zero out components	w[w <= 0.5] = 0	w = 0.0 0.0 0.0 0.0 0.0 0.0 0.6 0.7 0.8 0.9 1.0

19. In the first row, we need to start by remembering from the previous quiz what happened when we typed w less than or equal to 0.5. When we type that, we get a vector of logical values, true or false with the first six values being true and the last five values being false. We pass that vector of logical values inside square brackets. And the vector w, what happens is that we get the indices of w corresponding to locations where we have logical values true. since the vector w less than or equal to 0.5 have the first six values true, the first six indices will be retrieved and will get the vector 0.0, In the second row, we get the first outcome as the first row using slightly different syntax. We use the command subset and we pass the vector w holding the value 00.1.2 all the way to 1. And then we pass another vector of logical values containing true or false. The function extracts the elements of w corresponding to the positions where the second logical vector has true values. In the third row we assign the value 0 to a subset of the vector w. The subset of the vector w is defined by the condition w less than or equal to 0.5, which corresponds to the first six values being true. Therefore, the extracted first six values will be assigned the value 0, and the value of the vector after this command will be the same as the original vector w with the first six values zeroed out.



Creating Arrays Quiz

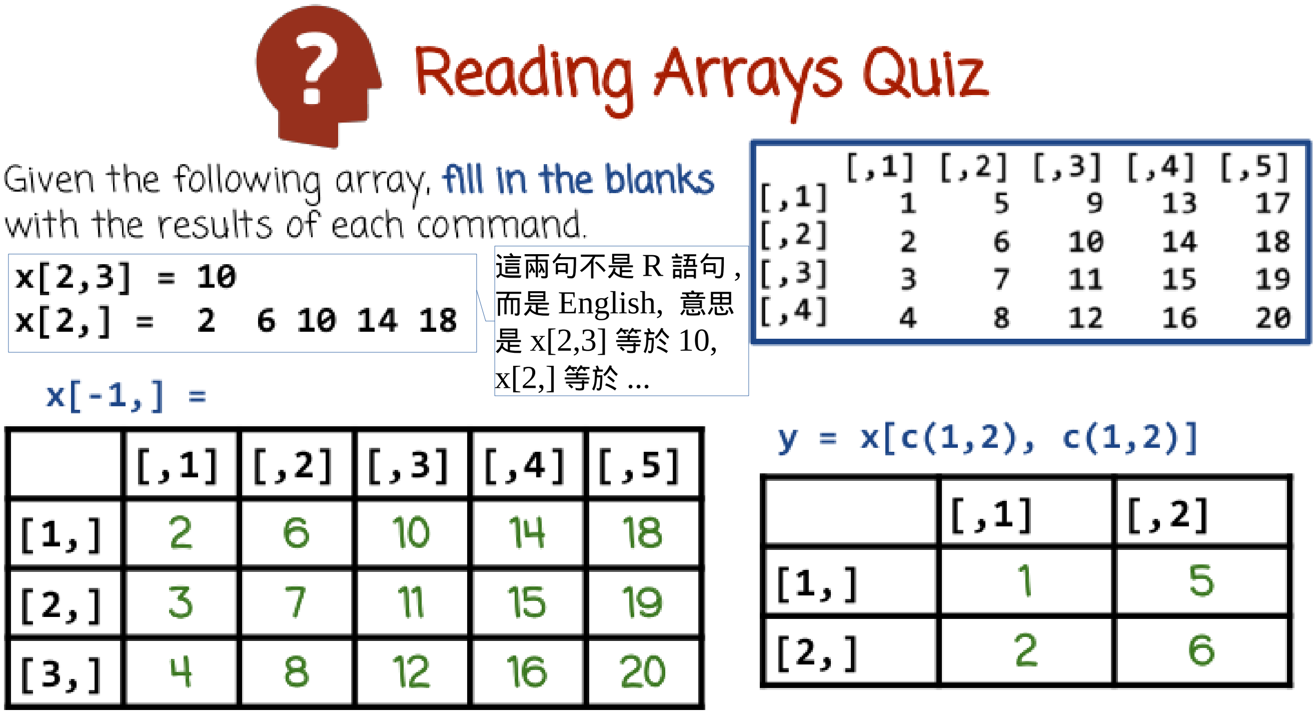
```
z = seq(1, 20,length.out = 20)
x = array(data = z, dim = c(4, 5))
```

Fill in the boxes with the values stored in the array.

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	1	5	9	13	17
[2,]	2	6	10	14	18
[3,]	3	7	11	15	19
[4,]	4	8	12	16	20

20. Arrays in R are a multi-dimensional generalization of vectors. They may have two or more dimensions. In the case of two dimensions, they correspond to tables. Here is a quiz where we type the first two lines here, which creates an array. Please fill in the boxes with the values stored in that array.

21. When we run the first command, what we get is a vector of values 1, 2, 3, 4, 5, 6 and so on until 20. If you remember the seq function, it creates a vector of lengths 20 equally spaced values between 1 and 20. So that gives us a vector of values. The second command takes the vector z and passes it to a function array(). The function array() takes two arguments in this case. The first argument is the vector of values and it assigns it to the argument data. The second is a vector of two values, 4 and 5. It assigns this vector of 4 and 5 to the argument dim, which is short for dimension. The function array will take the data in z, and reshape it to be an array of sizes 4 by 5.



Reading Arrays Quiz

Given the following array, fill in the blanks with the results of each command.

```
x[2,3] = 10
x[2,] =  2  6 10 14 18
```

這兩句不是 R 語句，而是 English，意思是 x[2,3] 等於 10, x[2,] 等於 ...

```
     [,1] [,2] [,3] [,4] [,5]
[,1]    1    5    9   13   17
[,2]    2    6   10   14   18
[,3]    3    7   11   15   19
[,4]    4    8   12   16   20
```

x[-1,] =

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	2	6	10	14	18
[2,]	3	7	11	15	19
[3,]	4	8	12	16	20

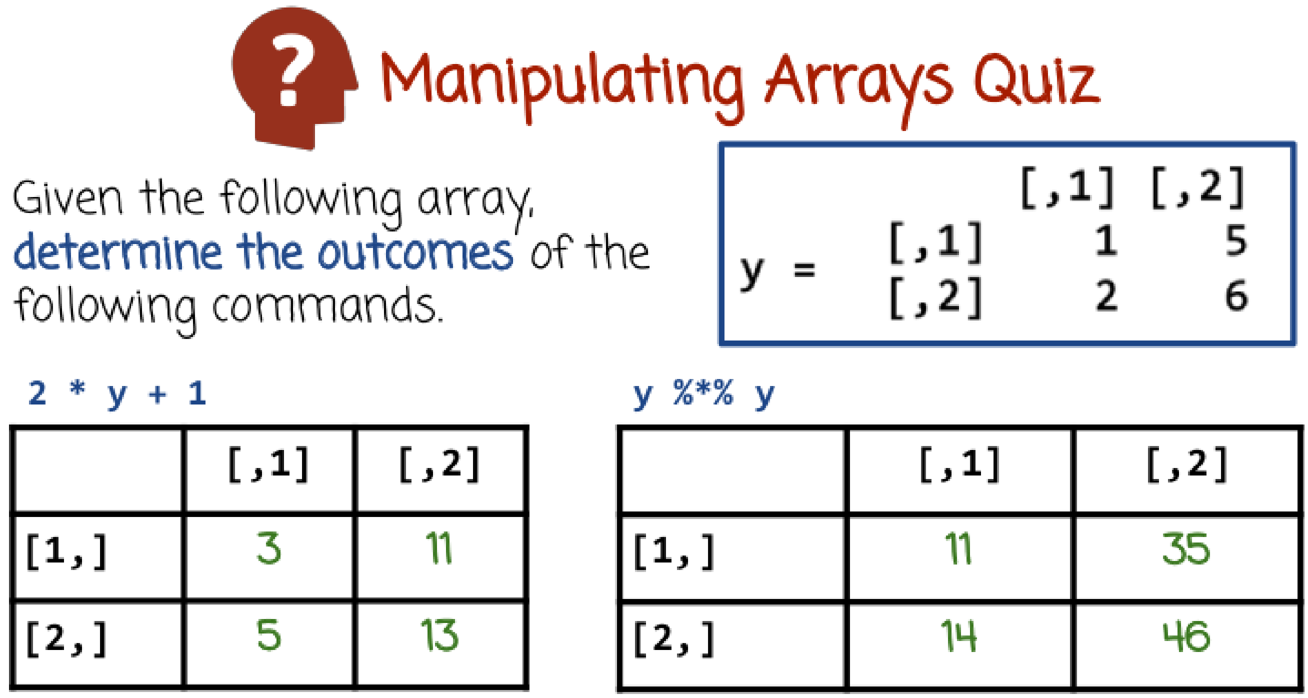
y = x[c(1,2), c(1,2)]

	[,1]	[,2]
[1,]	1	5
[2,]	2	6

22. In this quiz, we will work with this array right here, the array we created in the previous quiz of values of 1 to 20, sorted in an array of sizes 4 by 5. If we want to access a specific element in the array, we can do that using square brackets, just like we access elements of a vector. But because we have, in this case, two dimensions, we need to separate the two indices with a comma. The first index correspond to the row index. And the second one is the column index. If we leave either the row or the column index out, that corresponds to taking all possible values within that dimension. So in this case, if we write x [2,] we left the column dimension unspecified and the row dimension being 2. As a result, we'll get the entire second row. In the quiz, please look at the commands listed here and here and fill in the tables with the values that result from running these commands.

23. In this case, we have the array, x, with a row index, -1, and an unspecified column index. 左邊的: The negative row index actually corresponds to take all rows except for the specified row. So -1 will correspond to all rows except for the first row. Since we left the column unspecified, this corresponds to all the columns. The result being, we'll get a new array which corresponds to the old array, specifically, the second, the third, and the fourth rows of the old array. In this case, we refer to the array, x, but instead of referring to the array, x, with one pair or row column indices, 右邊的: we refer to the array, x, with a vector of rows and a vector of columns, specifically vector (1,2) of rows, and (1,2) of columns. Remember the function c concatenates the values inside of it and returns a vector with the corresponding values. In this case, we'll create a new array, y, that is extracted from the old array, x, specifically the first two rows and the first two columns.

24. In this quiz, we will manipulate arrays. Given the following array in the top right corner, determine the outcomes of the following commands. 2 * y + 1, y % * % y.



Manipulating Arrays Quiz

Given the following array, determine the outcomes of the following commands.

```
            [,1] [,2]
y =    [,1]    1    5
       [,2]    2    6
```

2 * y + 1

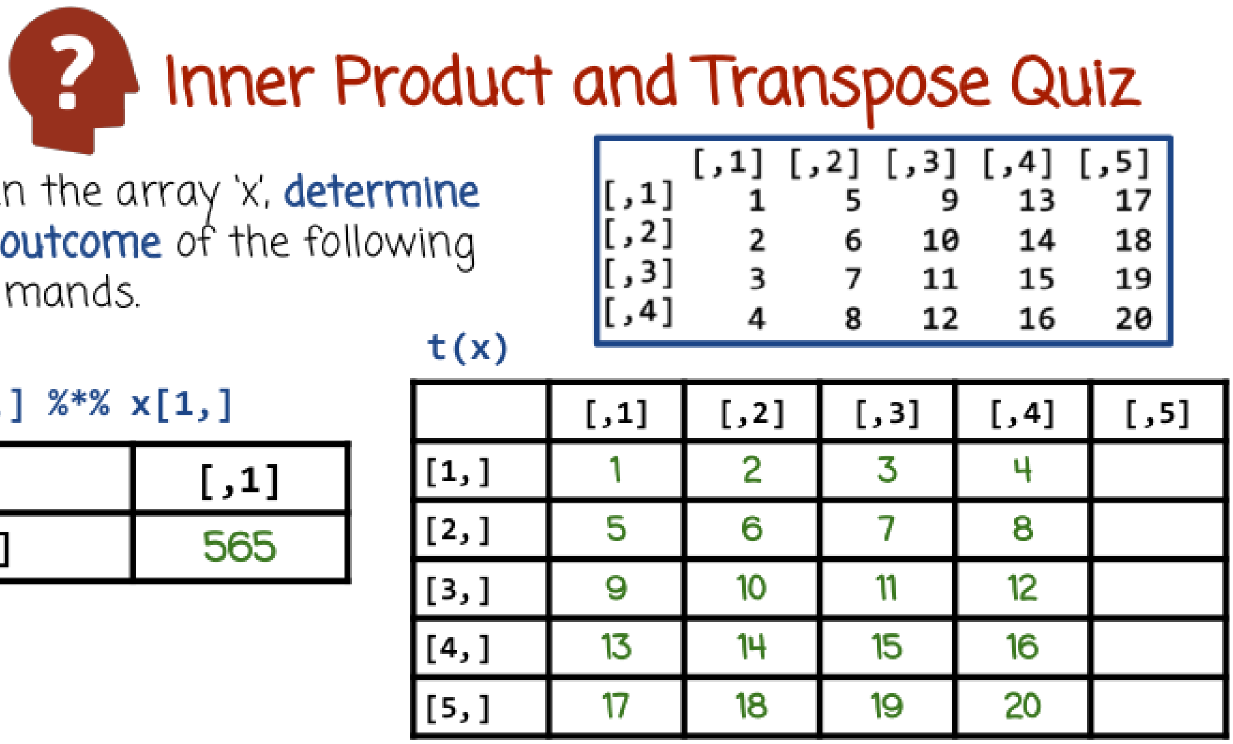
	[,1]	[,2]
[1,]	3	11
[2,]	5	13

y %*% y

	[,1]	[,2]
[1,]	11	35
[2,]	14	46

25. In this case, we have standard algebraic operators, such as multiplication and addition applied to an array. 左邊的: The arithmetic operations are applied to each elements of the array separately. And we get a new array, as a result, of a similar size. In this case, if we take every element of this array, and we multiply it by 2 and add 1, we get the resulting array here. Please note that this will also apply in the same way for vectors or for higher dimensional arrays, such as three-dimensional arrays and so on. 右邊的: In this case, we have a matrix multiplication operator (%*%, 就是線性代數中的矩陣相乘). The array y is interpreted as a matrix and we have the matrix y multiplied by itself using matrix multiplication operation, which is different from element y's product. For example, this element right here (11), of the result, is the result of multiplying the first row of y with the first column of y, using the inner product operation.

26. Given the array x, determine the outcome of the following commands. This command here on the left and this command here on the right.



Inner Product and Transpose Quiz

Given the array 'x', **determine the outcome** of the following commands.

```
     [,1] [,2] [,3] [,4] [,5]
[,1]    1    5    9   13   17
[,2]    2    6   10   14   18
[,3]    3    7   11   15   19
[,4]    4    8   12   16   20
```

```
x[1,] %*% x[1,]
```

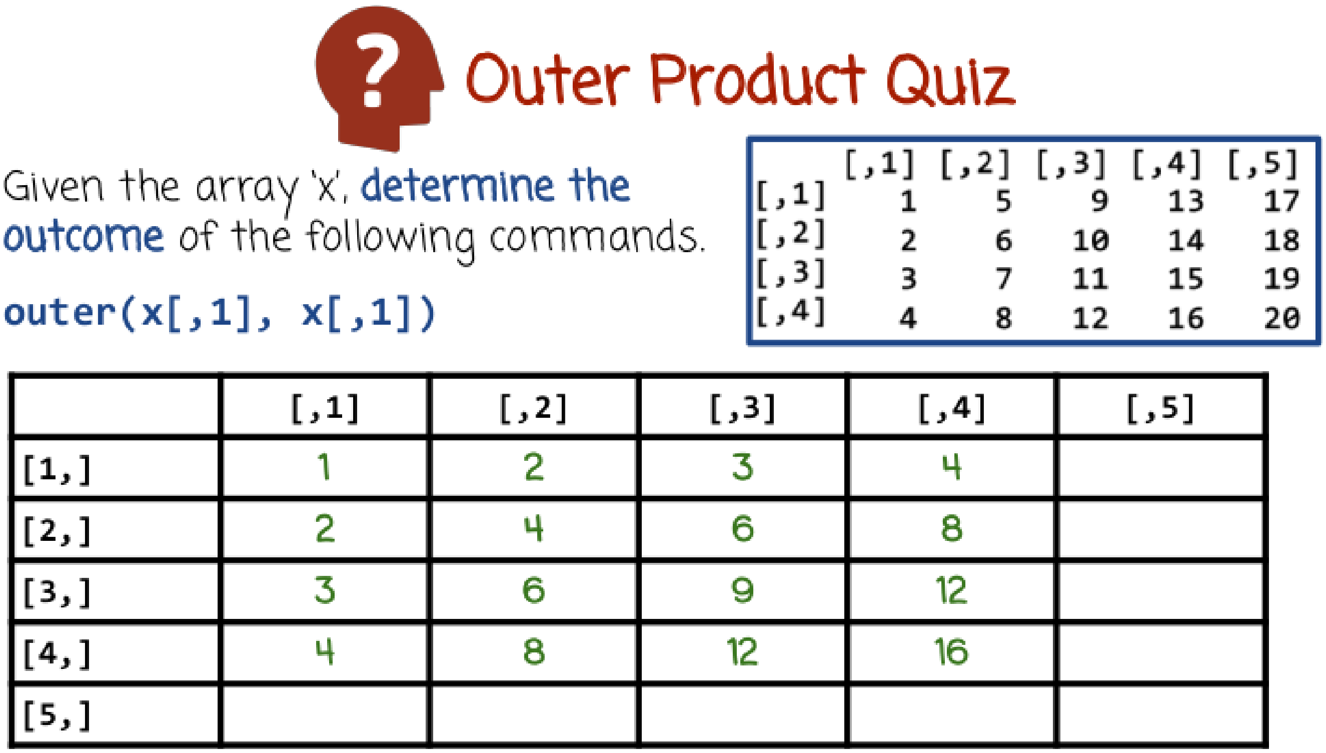
	[,1]
[1,]	565

```
t(x)
```

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	1	2	3	4	
[2,]	5	6	7	8	
[3,]	9	10	11	12	
[4,]	13	14	15	16	
[5,]	17	18	19	20	

27. In the first command we have the operator percent and duplication percent between two vectors. This operator (%*%) corresponds to the inner product operator between two vectors. The inner product operator multiplies the corresponding elements of the two vectors and sums them up. The result is 565. In the case of the second command, we have the function t() applied to the array x. The function t() transposes the matrix or array so that rows become columns and columns become rows. The four by five array is now a five by four array.

28. Given the array x, determine the outcome of the following command.



Outer Product Quiz

Given the array 'x', **determine the outcome** of the following commands.

```
outer(x[,1], x[,1])
```

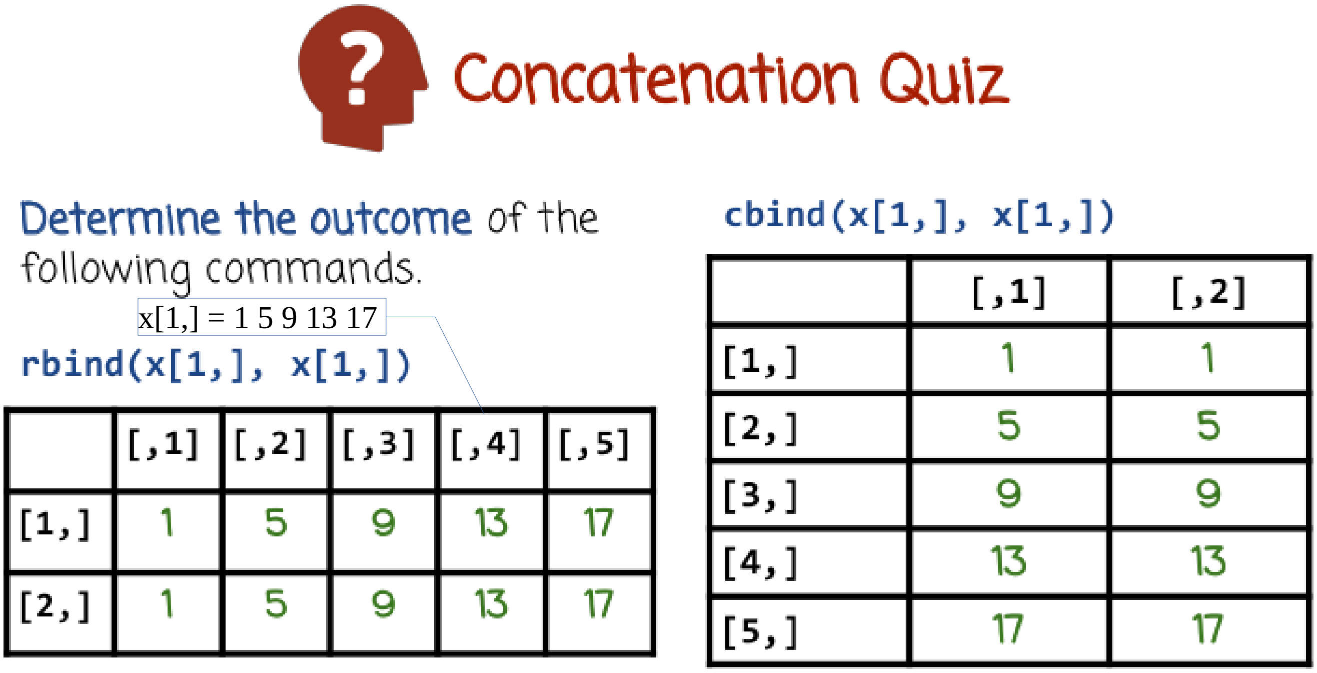
```
     [,1] [,2] [,3] [,4] [,5]
[,1]    1    5    9   13   17
[,2]    2    6   10   14   18
[,3]    3    7   11   15   19
[,4]    4    8   12   16   20
```

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	1	2	3	4	
[2,]	2	4	6	8	
[3,]	3	6	9	12	
[4,]	4	8	12	16	
[5,]					

vector 之間的 outer product 即 : A = outer(B, C), A_ij = B_i * C_j.

29. We have here a function outer apply to two vectors. The function outer performs the outer product between these two vectors right here. The outer product of two vectors is a matrix whose ij element is a product of the i component of the first vector and the j component of the second vector.

30. Determine the outcome of the following command where x is the array from the previous quiz.



Concatenation Quiz

Determine the outcome of the following commands.

x[1,] = 1 5 9 13 17

```
rbind(x[1,], x[1,])
```

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	1	5	9	13	17
[2,]	1	5	9	13	17

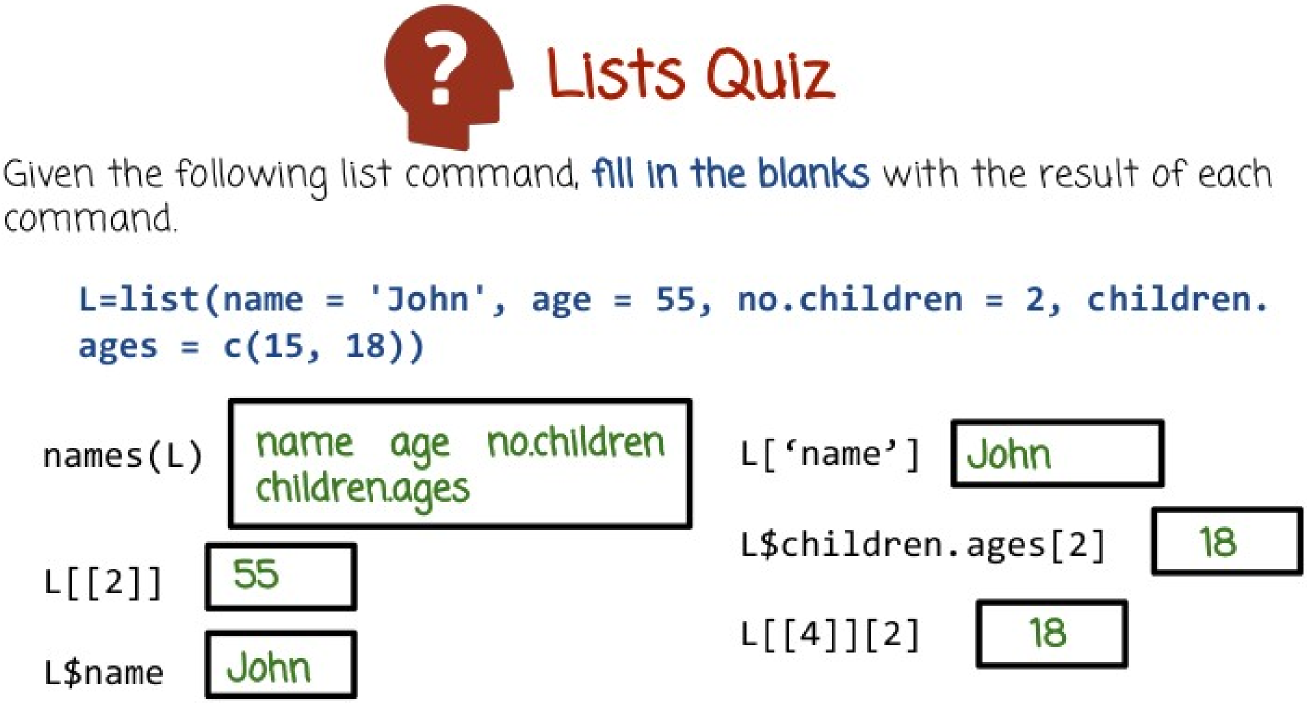
```
cbind(x[1,], x[1,])
```

	[,1]	[,2]
[1,]	1	1
[2,]	5	5
[3,]	9	9
[4,]	13	13
[5,]	17	17

31. We have the function rbind applied to two vectors. Rbind concatenate the two vectors as rows. R stands for row or rowbind.

32. Determine the outcome of the following command, where x is the array from the previous quiz.

(圖為上圖)
33. We have the function cbind. Cbind concatenates the two vectors as columns.



Lists Quiz

Given the following list command, fill in the blanks with the result of each command.

```
L=list(name = 'John', age = 55, no.children = 2, children.
ages = c(15, 18))
```

names(L) name age no.children children.ages

L[[2]] 55

L$name John

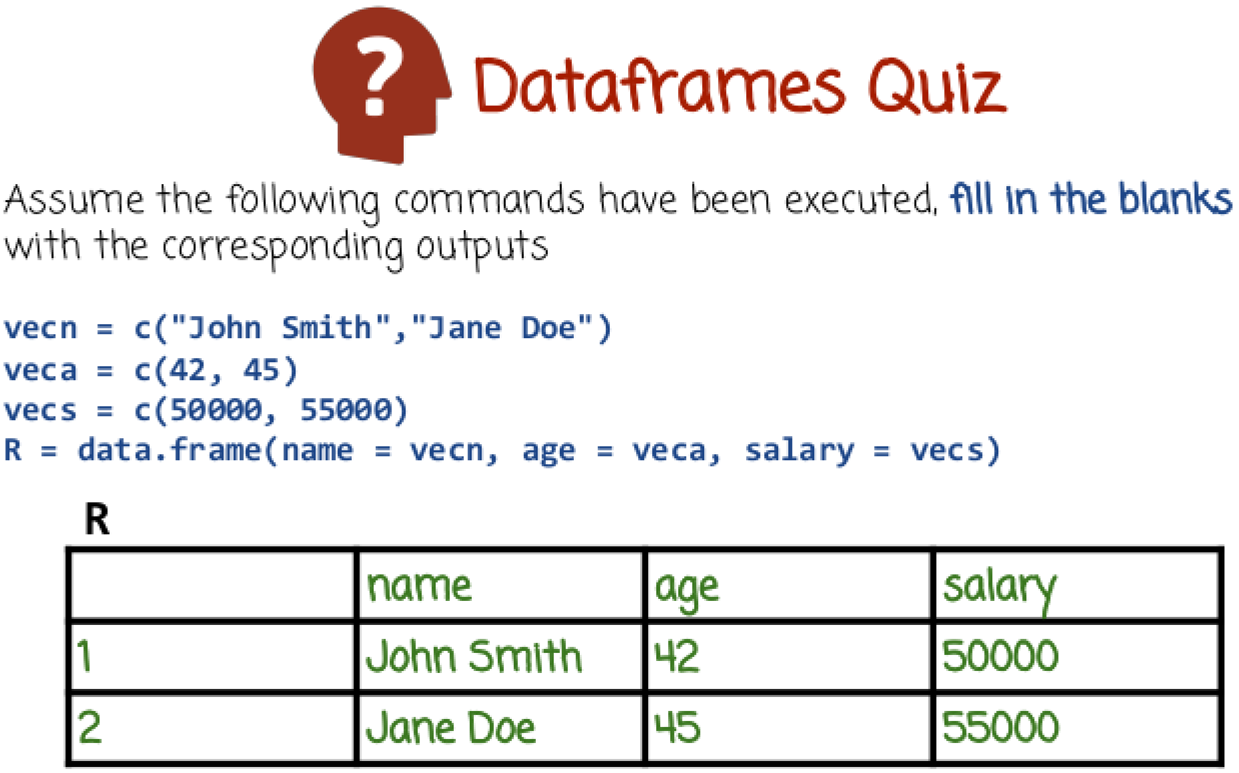
L['name'] John

L$children.ages[2] 18

L[[4]][2] 18

34. Lists in R are ordered collections of possibly different types. So we can have a list holding in the first place a string, in the second place a number, the third place a Boolean, etc. Often times in lists we assign names to different positions or elements so that we can refer to the elements using their name, rather than the index or position. Here is an example. We have here a list where we assign the value John to the first position and give it a name, name. We assign it a value 55 to the second position And assign it the name age. We assign the value 2 to the third position. And assign it the name and no.children and so on. In the last case we assign a vector (15, 18) to the last position and assign it the name children.ages. You see each element here may have a different type. A string, a number, a number, and then a vector. In this quiz, given the following command, fill in the blanks with the results of each command.

(圖為上圖)
35. names() is a function that returns the names of the positions. In this case (L[[2]]) we have reference to second position of a list but we have this double square bracket notation. Double square bracket notation in lists, returns the object in the second position. In this case it's 55. If we use only a single brackets notation, we would get a list containing the object 55. If we want instead of a list containing 55, the actual linear object meaning the value of 55 we need to use the double square bracket notation. In this case, we refer to one of the list elements using its name. In this case (L$name) the name is name. The reference is done using the $ operator, so we refer to the element of the list that has the name name, in this case, it's the first element, and we get back the value of that object. This case (L['name']), we also refer to a specific object in the list using its name, when we use a sightly different syntax similar to the syntax in python of accessing hash map or hash table, the outcome of this command and this command are the same and you can use either format depending on which one you find more comfortable. In this case we refer to the value or object in the list having the name children.ages. This is the last element. The value of that object is a vector, But because we have the square brackets at the end with the value 2. We're going to extract the second value of that vector and return 18. The last case we get exactly the same result, as the second to last case. But instead of referring to the fourth object by name, we refer to it by position.



Dataframes Quiz

Assume the following commands have been executed, fill in the blanks with the corresponding outputs

```
vecn = c("John Smith","Jane Doe")
veca = c(42, 45)
vecs = c(50000, 55000)
R = data.frame(name = vecn, age = veca, salary = vecs)
```

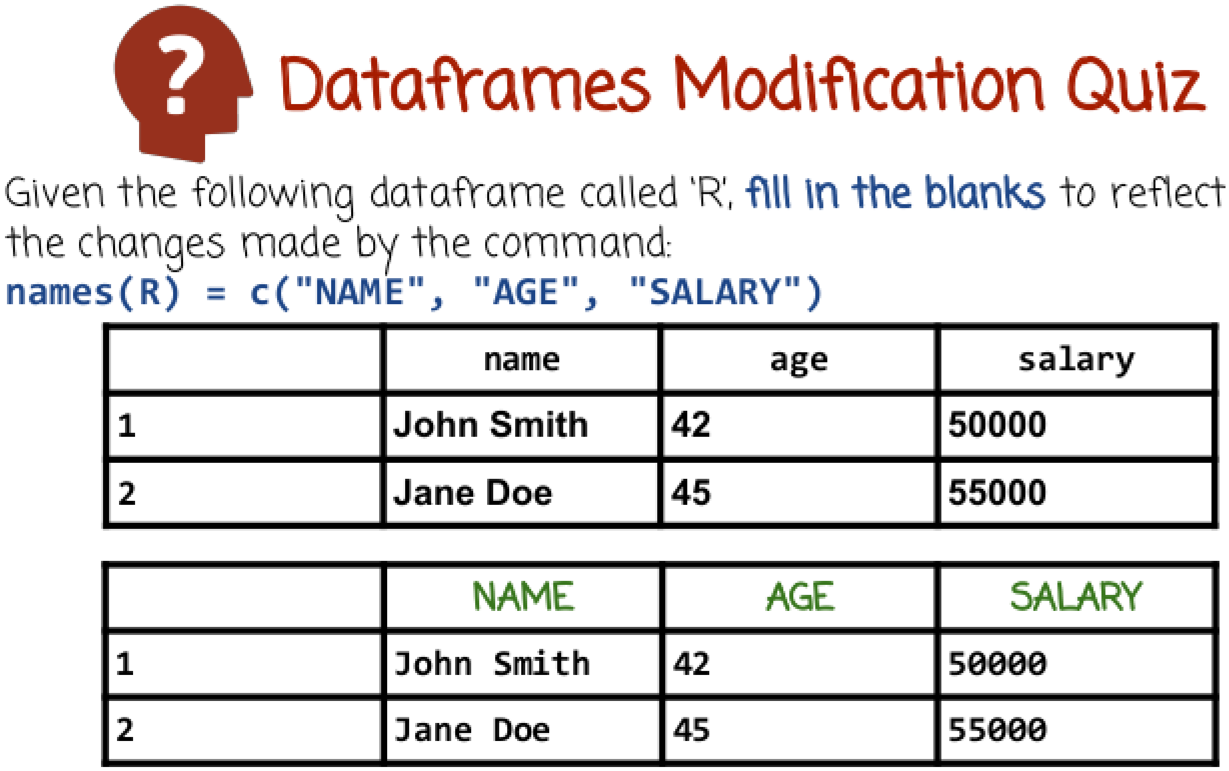
R

	name	age	salary
1	John Smith	42	50000
2	Jane Doe	45	55000

36. Dataframes in R are ordered sequence of lists sharing the same signature or same data types. So for example, we can have a list with a specific types of variables (該 list 內部的元素都為同一個 type), and then another list with the same types (該 list 內部的元素都為同一個 type), and if we think of these lists as being rows In a table, the table would be a data frame. A popular use case for data frame is a table where the rows correspond to data examples, or instances and the columns correspond to dimensions or features In this quiz, assuming the following commands have been executed, fill in the blanks with the corresponding output.

(圖為上圖)
37. In the first command here, we create a vector of strings, having the two strings John Smith and Jane Doe. In the second function, we create a vector of numbers 42, 45. That would later correspond to the ages of the individuals whose names is specified in the first vector. In the third command we create a vector of numbers 50,000 and 55,000 corresponding to the salary of the individuals whose names are in the first vector. So we have three vectors, vecn, veca, and vecs. We're going to create the dataframe using the data.frame function and we're going to pass the three vectors to the data.frame function. We also going to specify which names we're going to use to refer to these columns inside a data frame. So after we execute the data of the frame function with the corresponding arguments, we get the table or data frame here having three different columns, name, age, salary in two rows. The two rows hold the following values, John Smith, 42, 50,000. The second one Jane Doe, 45, 55,000. As you can see in this data frame, we have two rows. Each row can be thought of as a list with three different values having the same signature, the first one being a string and the second and the third one being numbers.

38. Given the following dataframe, called R, fill in the blanks to reflect the changes made by the command written here.



Dataframes Modification Quiz

Given the following dataframe called 'R', fill in the blanks to reflect the changes made by the command:

```
names(R) = c("NAME", "AGE", "SALARY")
```

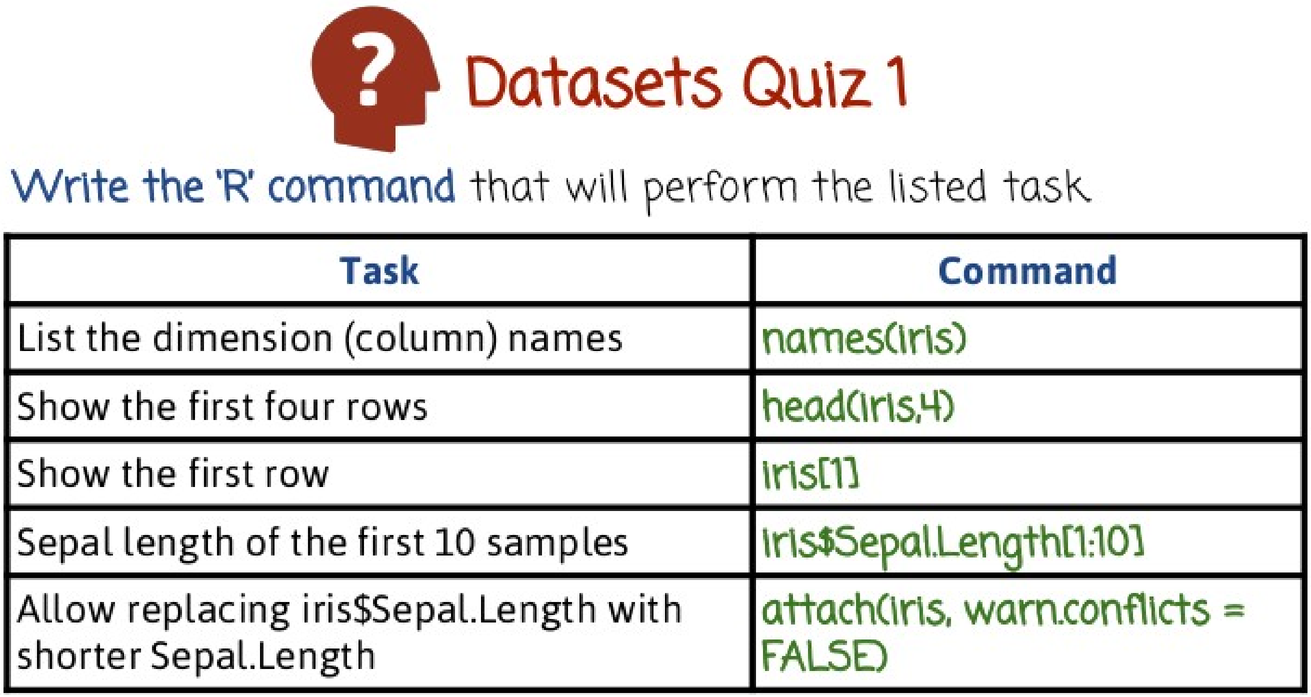
	name	age	salary
1	John Smith	42	50000
2	Jane Doe	45	55000

	NAME	AGE	SALARY
1	John Smith	42	50000
2	Jane Doe	45	55000

39. The command here creates a vector of three strings NAME, AGE, SALARY in all caps, and assigns them from what is returned from names(R). names(R) returns the names of the columns of the dataframe R. As a result of this command right here, we are basically going to replace the previous names, name, age, salary, lower caps with NAME, AGE, SALARY, upper caps.

The next two quizzes refer to the Iris Dataset.
You can load it using the R command .... data(Iris). (在我電腦上好像不用 load, 就可以直接用 Iris)
(啟動 R 後, 在 terminal 中打 iris, 就可以顯示所有數据)

To see all the datasets that are available in R, go to:
The R Datasets
https://stat.ethz.ch/R-manual/R-devel/library/datasets/html/00Index.html



Datasets Quiz 1

Write the 'R' command that will perform the listed task

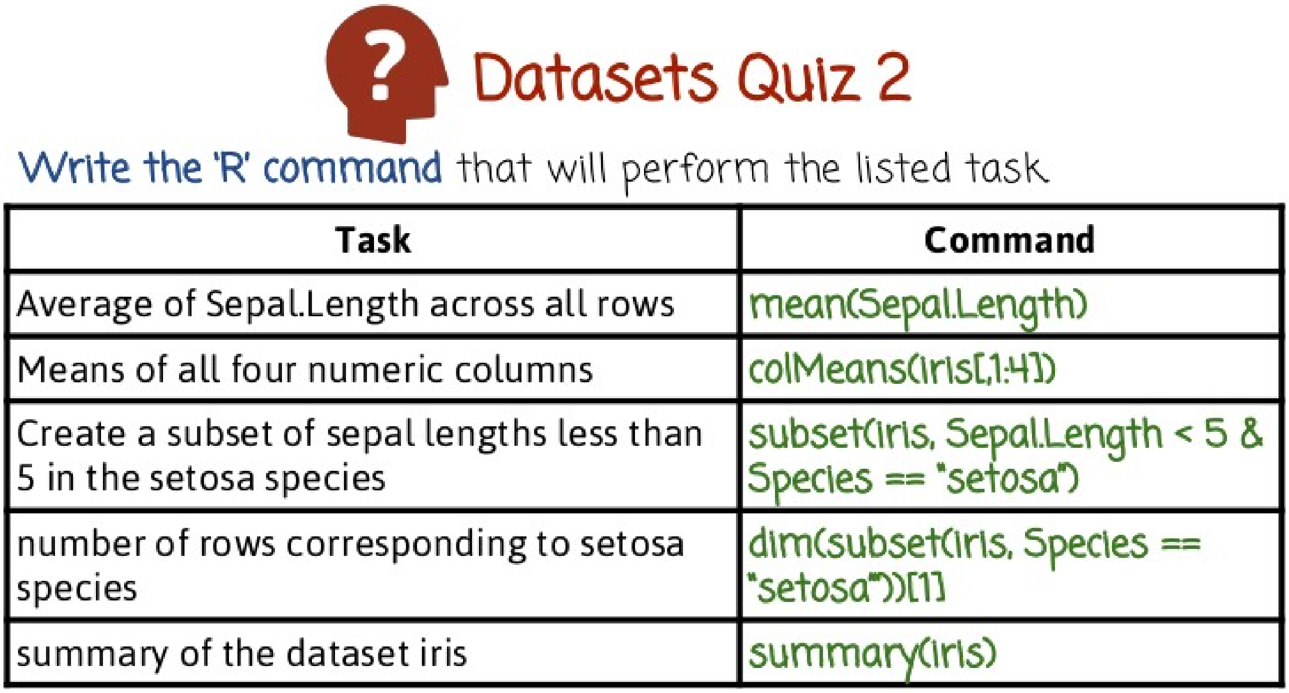
Task	Command
List the dimension (column) names	names(Iris)
Show the first four rows	head(Iris,4)
Show the first row	Iris[1]
Sepal length of the first 10 samples	Iris$Sepal.Length[1:10]
Allow replacing iris$Sepal.Length with shorter Sepal.Length	attach(Iris, warn.conflicts = FALSE)

上表中的iris第一個字母i要小寫.

40. In this quiz we're going to use a famous data set called Iris. Iris data set have many measurements of different flowers. For each flower we're going to consider three different measurements. The length of the width of the sepal and the length of the width of the petal. And we're also going to consider the species of the flower. There are three specific species, Iris setosa, versicolor and virginica. So the data frame has five columns, four of them are numeric having measurements of flower length and width and the last one Is a string holding the name of the species. You can find more information on the data set by typing help Iris. In this quiz, write the R commands that will perform the listed tasks.

41. The first case we need to issue a command that will list the dimension names. As we saw before, the function names() accomplishes that task (返回所有列的名字: "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width" "Species"). In the second case we want to show the first four rows. We can do that using the function head. And we also pass a second parameter, 4, to specify how many rows we want to display. We can also use a different function to display the first four rows, but head is a pretty convenient one. In the third case we want to show the first row. Well, the first option is to just use head(iris,1). But here is another example iris[1] (但我電腦上返回的是前兩列). If we have a data frame and we use square brackets with one number in the middle: 1, we'll basically access the first row. In the fourth case, we want the sepal length of the first 10 samples. We refer to the sepal length column using the name notation. So we use the $ operator followed by the name of the corresponding column. And then we need to apply the index 1:10 to tell R to return only the first 10 values. (last case) Normally we need to refer to the sepal length column using iris$Sepal.Length syntax. In some cases we may want to use a shorter syntax, just Sepal.Length, without the iris$. We can do that by issuing the function attach.

42. Write the R command that will perform the listed task.

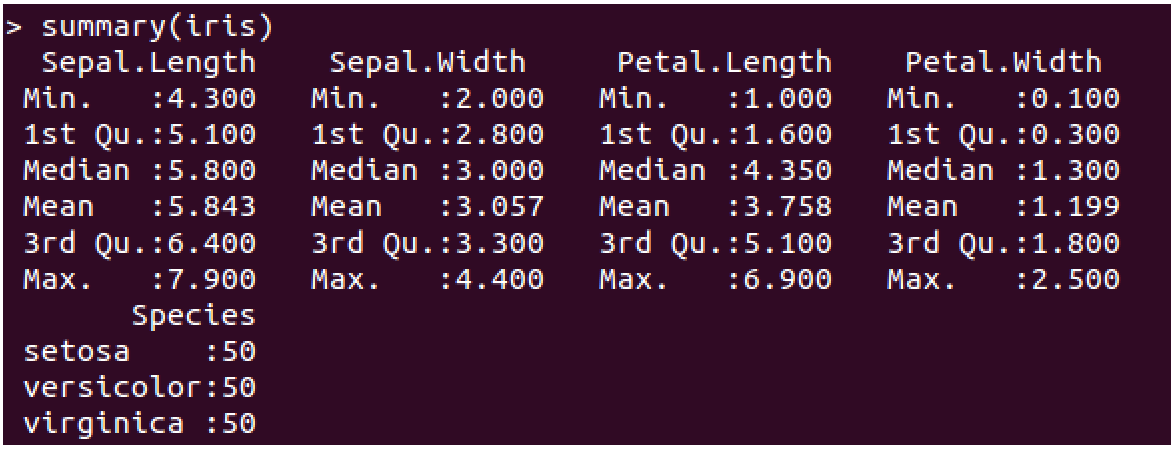


Datasets Quiz 2

Write the 'R' command that will perform the listed task.

Task	Command
Average of Sepal.Length across all rows	mean(Sepal.Length)
Means of all four numeric columns	colMeans(iris[,1:4])
Create a subset of sepal lengths less than 5 in the setosa species	subset(iris, Sepal.Length < 5 & Species == "setosa")
number of rows corresponding to setosa species	dim(subset(iris, Species == "setosa"))[1]
summary of the dataset iris	summary(iris)

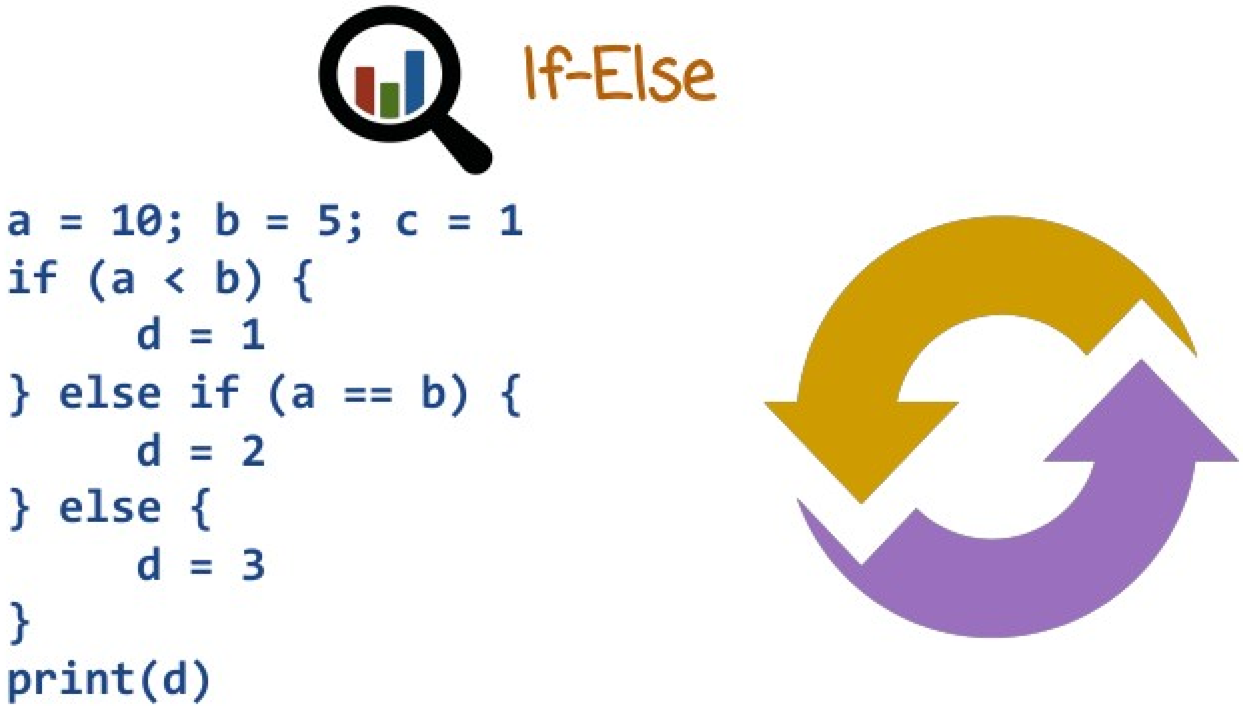
我電腦上運行 summary(iris)之結果(terminal 是放到最大的):



```
> summary(iris)
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width   
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100  
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300  
 Median :5.800   Median :3.000   Median :4.350   Median :1.300  
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199  
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800  
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500  
       Species  
 setosa    :50  
 versicolor:50  
 virginica :50  
```

43. In the first case we want to compute the average of the Sepal.Length column across all rows. We can do that using the function mean. Then we apply iris$Sepal.Length. In this case we want to compute the means of all numeric columns. You can do that using the function colMeans. We pass the data from to that, but only the 4 columns, because the last column has a categorical variable. We can not compute it's mean. In this case we want to create a subset of sepal.length less than 5 within the setosa species. We can do that using the subset function. We first pass the data frame name and then the logical condition, in this case, Sepal.Length value has to be less than five and the Species column have to be equal to setosa. In this case, we want to measure the number of rows corresponding to the setosa species. One way to do that is use the subset function that we saw previously, where we basically extract all rows of the Iris data frame whose Species value corresponds to setosa, and then we apply the dim function to that, dim returns dimensions. But since this is a data frame, the dim function will return a vector of two values, the number of rows and the number of columns, we just want the

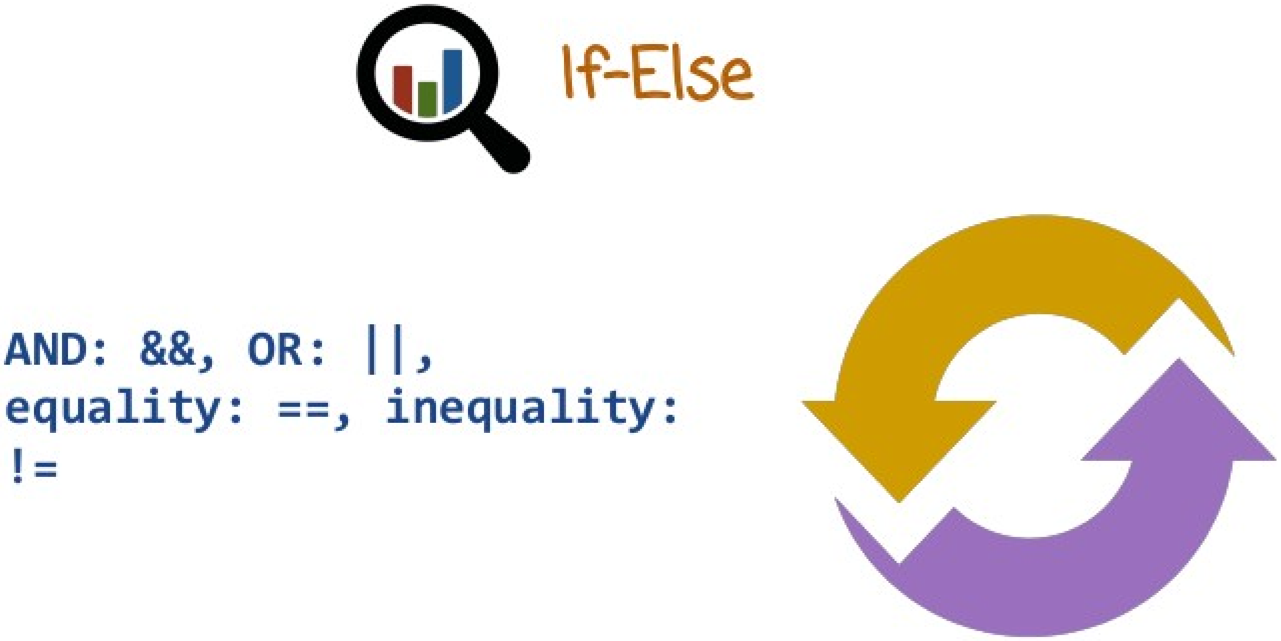
number of rows. So we use a square bracket here with the value 1. Telling R we just won the first value of that vector. In this case we want the summary of the data set. We can do that using a very convenient summary function that gives a summary of every data frame column. If it's numeric we get numbers such as the mean, median, quartiles, etc. And if it's categorical, we get a histogram of how many times different values appear.



If-Else

```
a = 10; b = 5; c = 1
if (a < b) {
      d = 1
} else if (a == b) {
      d = 2
} else {
      d = 3
}
print(d)
```

44. Control flow using if-else in R is pretty similar to other languages. Here is an example. We first have an if clause followed by else if that is executed, if the first if clause is not correct. We can have multiple else if clauses. At the end we have an optional else clause that is executed, if none of the previous clauses are executed.

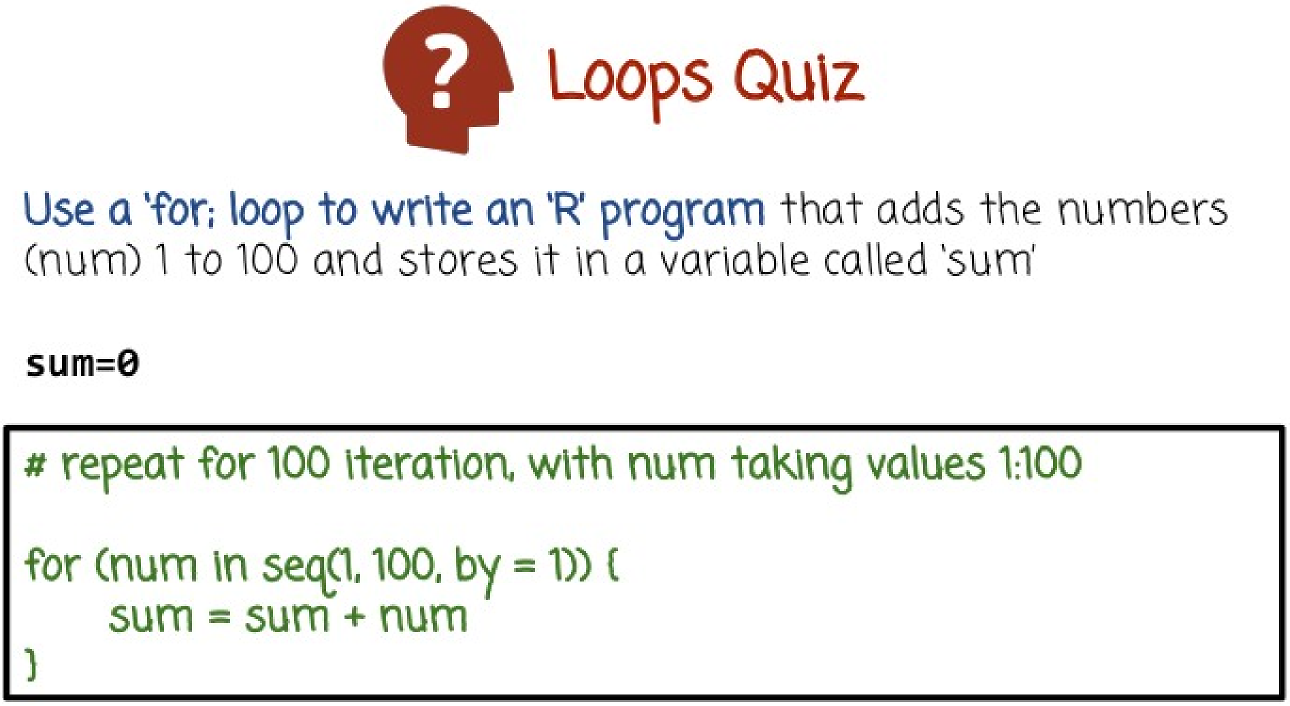


If-Else

```
AND: &&, OR: ||,
equality: ==, inequality:
!=
```

Logical operators in R are similar to the logical operators in C++ or Java. An AND operator is basically &&, OR operator is ||, equality is = =, and inequality is !=. These operators take two values and return true or false based on the logical condition that they describe.

45. In this quiz, use a for loop to write an R program that adds the numbers 1 to 100 and store it in a variable called sum. We're going to refer to the iteration variable, using the variable name num.



Loops Quiz

Use a 'for; loop to write an 'R' program that adds the numbers (num) 1 to 100 and stores it in a variable called 'sum'

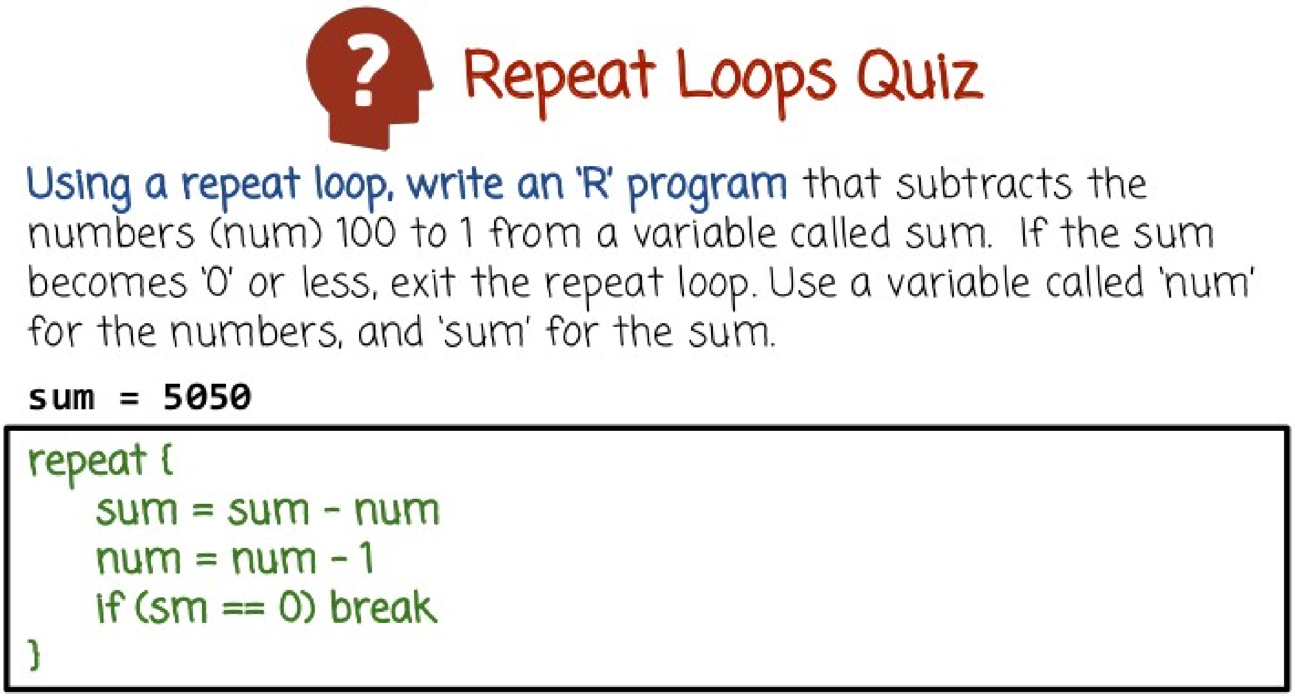
```
sum=0
```

```
# repeat for 100 iteration, with num taking values 1:100

for (num in seq(1, 100, by = 1)) {
     sum = sum + num
}
```

也可以寫為 for(i in 1:100)這樣的.

46. We have a for loop that iterates over every element of that vector here. The vector is seq(1, 100, by=1), which is a vector of values between 1 and 100 separated by distance of 1. That's basically 1, 2, 3, 4, all the way to 100. So the for loop with iterate over every value in that vector, and every iteration, the corresponding value will be assigned to the iteration variable num. Within the for loop, we're going to increment the variable sum with the iteration variable num.

47. Using a repeat loop write an "R" program that subtracts the numbers 100 to 1 using an iteration variable num from a variable called sum. If the sum becomes zero or less, exit the repeat loop. Use a variable called num for the numbers and sum for the sum.



Repeat Loops Quiz

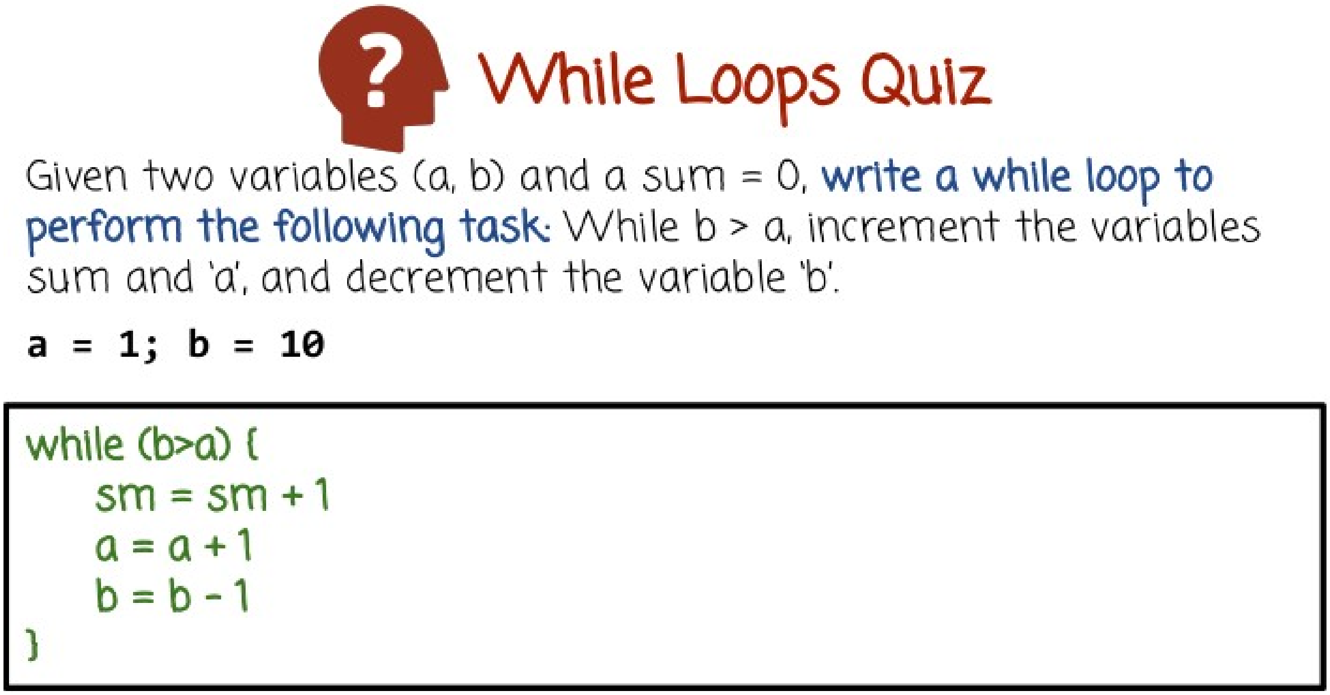
Using a repeat loop, write an 'R' program that subtracts the numbers (num) 100 to 1 from a variable called sum. If the sum becomes '0' or less, exit the repeat loop. Use a variable called 'num' for the numbers, and 'sum' for the sum.

```
sum = 5050
repeat {
    sum = sum - num
    num = num - 1
    if (sm == 0) break
}
```

A repeat loop must use a break statement to exit the loop.

48. We have here a repeat loop, that basically, gets repeated until we encounter the break command right here. Within the repeat loop, we subtract the value num from the variable sum. And then we decrement num by 1. And the final line is the condition that breaks out of the repeat loop, in the case that sum = 0.

49. Given two variables, a and b, and a sum = 0, write a while loop to perform the following task, while b is greater than a, increment the variables sum and a, and decrement the variable b.



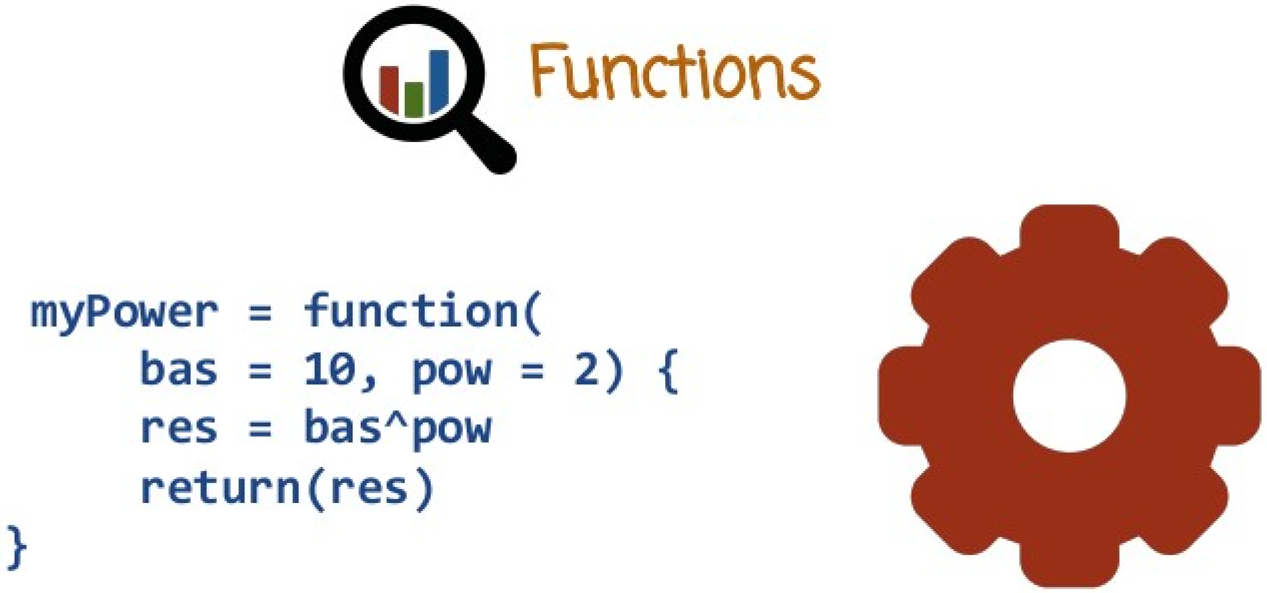
While Loops Quiz

Given two variables (a, b) and a sum = 0, write a while loop to perform the following task: While b > a, increment the variables sum and 'a', and decrement the variable 'b'.

```
a = 1; b = 10
```

```
while (b>a) {
    sm = sm + 1
    a = a + 1
    b = b - 1
}
```

50. Here we have a y loop that's executed as long as b is greater than a. Within the loop, we had increment the sm by 1. We increment a by 1, and decrement b by 1.

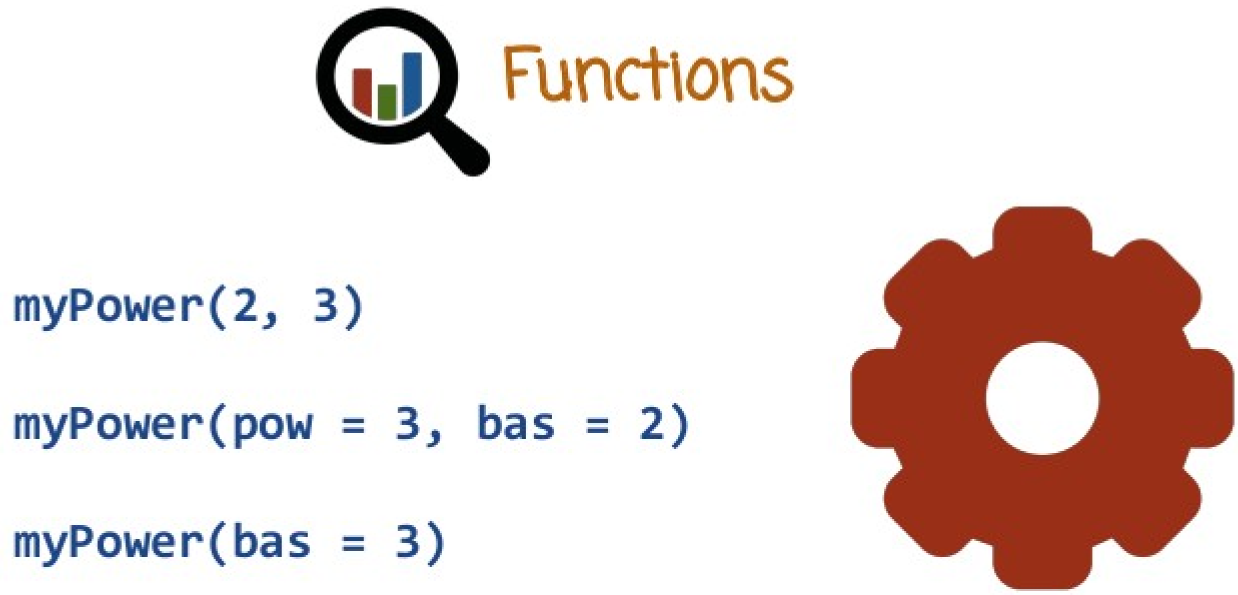


Functions

```
 myPower = function(
      bas = 10, pow = 2) {
      res = bas^pow
      return(res)
}
```

51. Let's see how to define a function in R. We define a function here called myPower. We do that by calling the function, function. And that function returns the function and assigns it to the name myPower. The result of this computation here is that we define the function myPower that can be later called by R in a different place. Let's see how the definition works. After the key word function, we provide the arguments. In this case, we have two arguments, bas and pow. The function will raise the first argument to the power of the second. Or, bas to the power of pow. We provide here default values 10 and 2 that will be assigned if the corresponding values are missing when the function is called. After

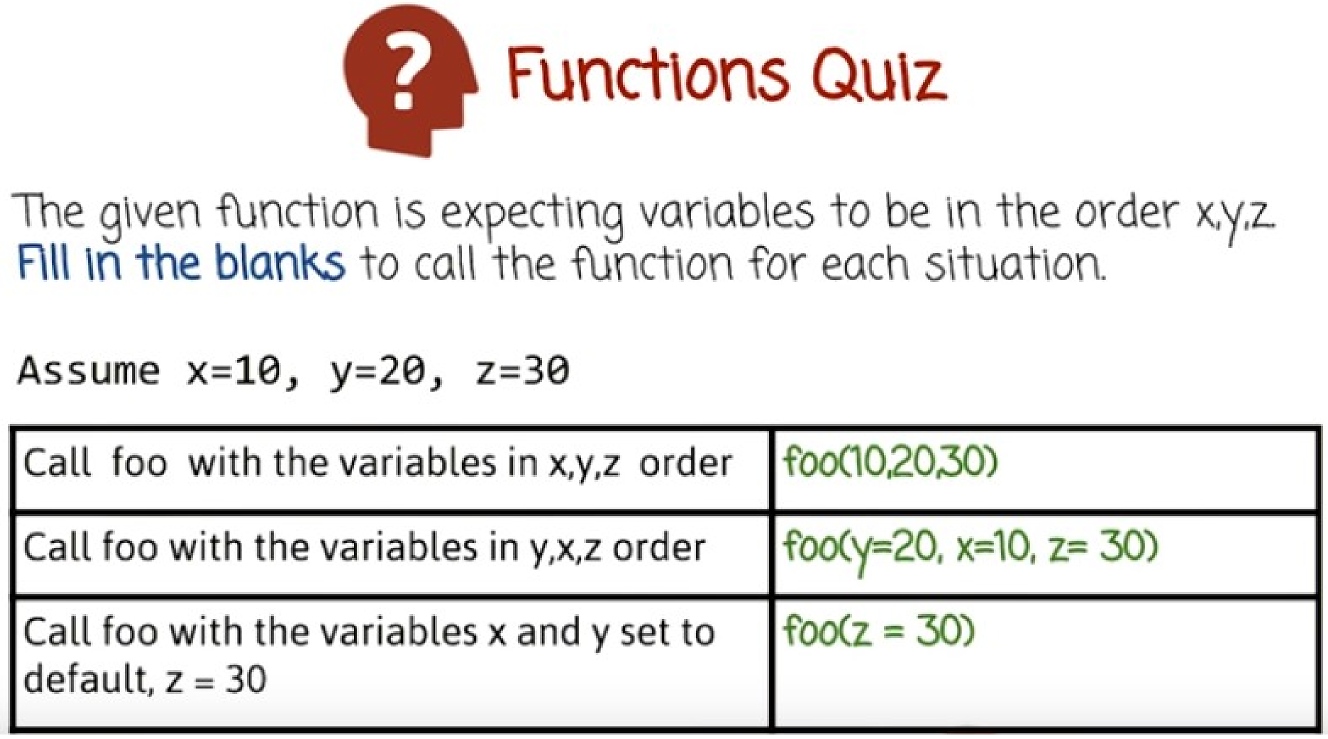
we provide the parameter names, we have curly braces and inside the curly braces, we have the function code. The function code raises bas to the power pow and then return the result.



Functions

```
myPower(2, 3)

myPower(pow = 3, bas = 2)

myPower(bas = 3)
```

Let's see some examples how to call a function. In the first case we call the function with two values two and three. These values are assigned to bas and pow in the corresponding order. In the second case we reverse the order (of the arguments). And because we reverse the order, we need to tell the function myPower that 3 should be assigned to the variable pow, and 2 should be assigned to the variable bas. In the third case we only pass one parameter, bas, and the function will assign the default value to the other parameter.



Functions Quiz

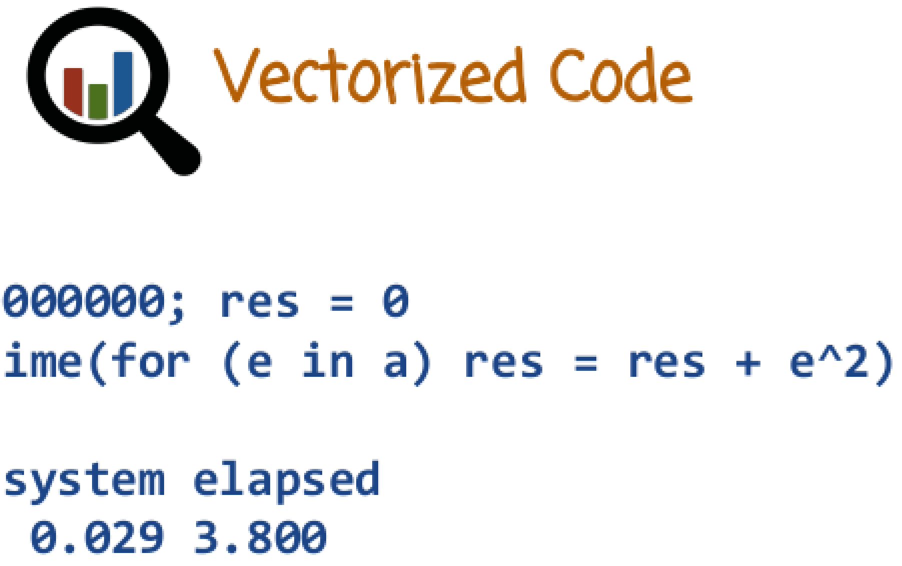
The given function is expecting variables to be in the order x,y,z. **Fill in the blanks** to call the function for each situation.

```
Assume x=10, y=20, z=30
```

Call foo with the variables in x,y,z order	foo(10,20,30)
Call foo with the variables in y,x,z order	foo(y=20, x=10, z= 30)
Call foo with the variables x and y set to default, z = 30	foo(z = 30)

52. When we have a function in R, we can pass arguments in the default order in which they're specified in the function definition. But if we provide names associated with these parameters, we can change the order of the parameters, and we can only pass a subset of the parameters potentially out of order. In this quiz, let's call this function foo with a variable x,y,z in that order. In the order y,x,z. And only with the variable z with x and y set to default. The past values should be 10, 20, and 30, assigned to the corresponding variables, x, y, and z.

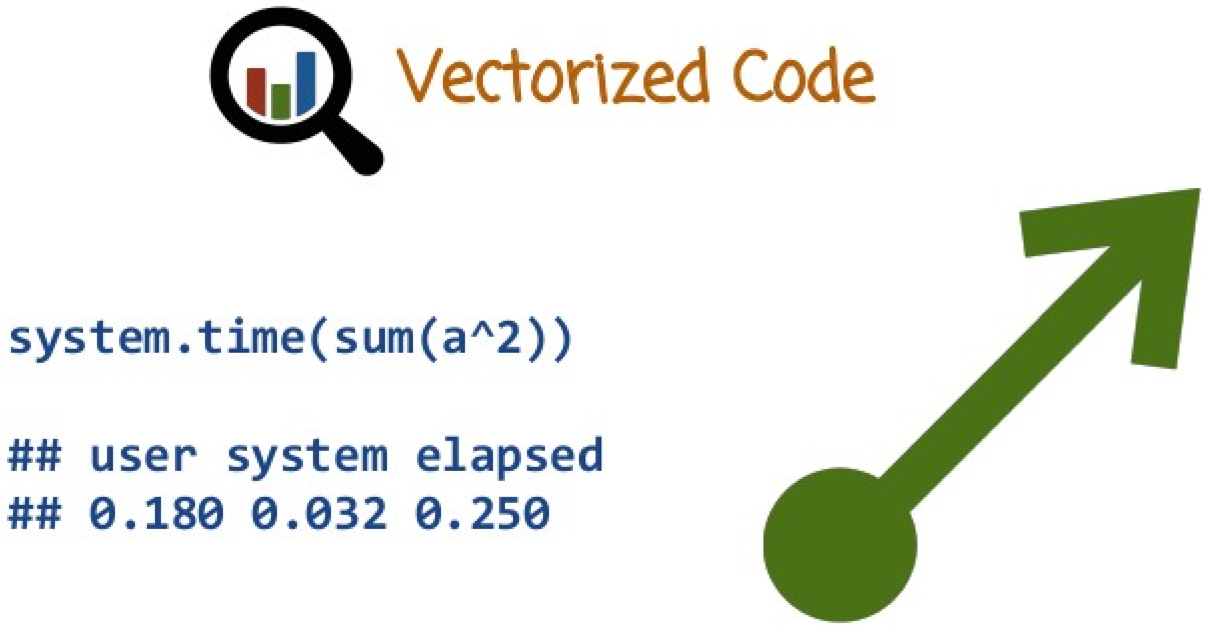
53. In the first case, we want to call foo with the variables 10, 20, 30 passed in that order. We don't need to provide any names because we just passed the parameters in the default order specified in the function definition. In the second case, we want to pass this values in a different order, so we have to specify names of the variables that these values will be assigned to. In the third case we only passed the value 30. Implying to the variable Z, we omit the variables X and Y because they're going to get the default value.



Vectorized Code

```
a = 1:10000000; res = 0
system.time(for (e in a) res = res + e^2)

## user system elapsed
## 3.742 0.029 3.800
```

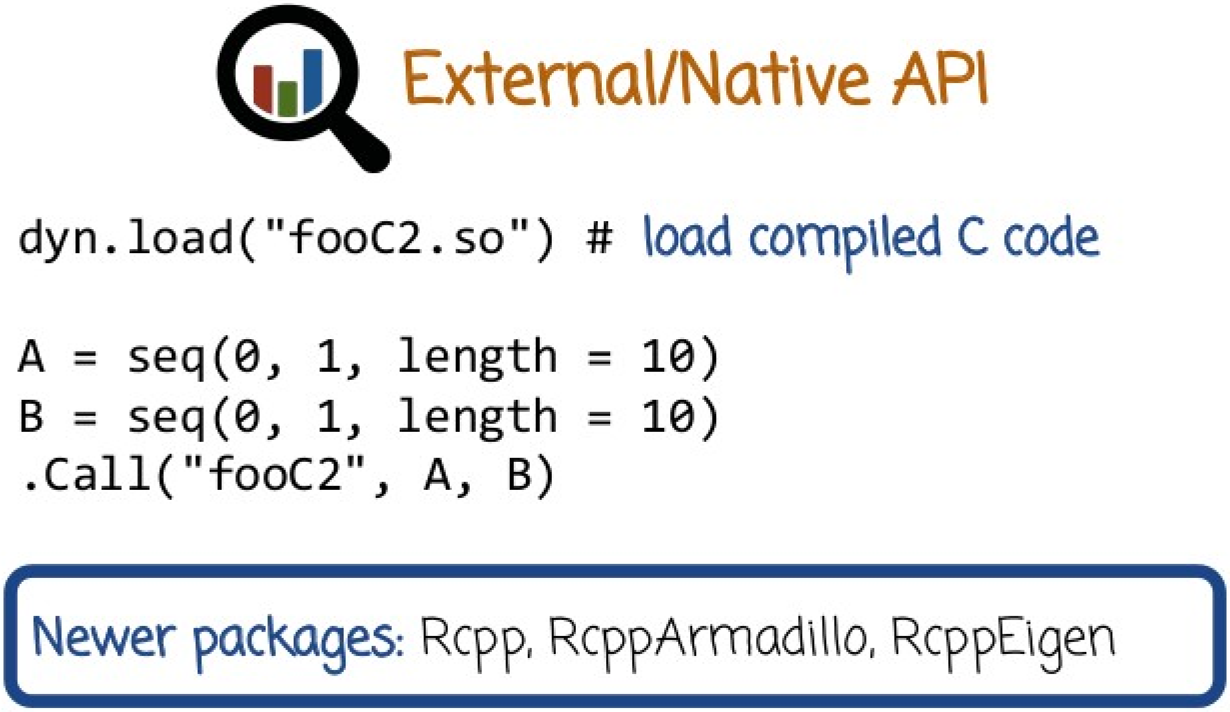
54. One of the disadvantages of R is that it can run slow, especially if the code has many loops or iterations. The reason for that is that R code runs inside an interpreter usually, and the interpreter has extra overhead. One way to go around that problem is to write vectorized code. Vectorized code means that instead of writing a loop, we're going to execute the same functionality without having a loop or loops in our code. This can cause a dramatic acceleration. Let's see an example. Here we create a big vector. And the code right here iterates over all elements within that vector. We're squaring each element and we're accumulating the values, putting the result in the variable res. We wrap this entire code in the function system.time, which returns the time it takes to compute that command. Because we have an explicit loop here, this code may be quite slow.



Vectorized Code

```
system.time(sum(a^2))

## user system elapsed
## 0.180 0.032 0.250
```

In this case, we have vectorized command that accomplishes the same functionality as the previous slide. Instead of having a loop, we raise every element of the vector a to the power 2, and then we accumulate the values in that vector. This is done using vectorized code. And you can compare the numbers here with the numbers in the previous slide and see the acceleration. The acceleration can become even more dramatic if the vector is longer or if you have nested loops.



External/Native API

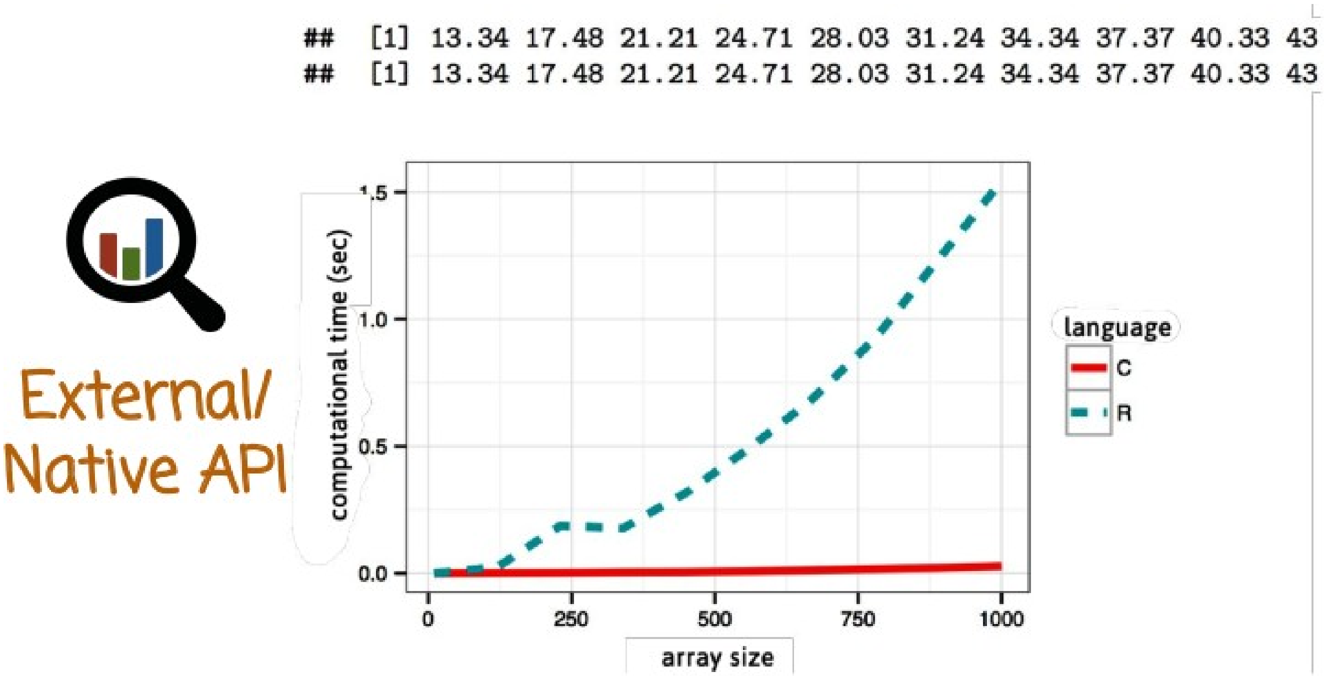
```
dyn.load("fooC2.so") # load compiled C code

A = seq(0, 1, length = 10)
B = seq(0, 1, length = 10)
.Call("fooC2", A, B)
```

Newer packages: Rcpp, RcppArmadillo, RcppEigen

External Native API
More information on this topic can be found in the recommended reading: R Programming

https://s3.amazonaws.com/content.udacity-data.com/courses/gt-cse6242/recommended+reading/R.pdf

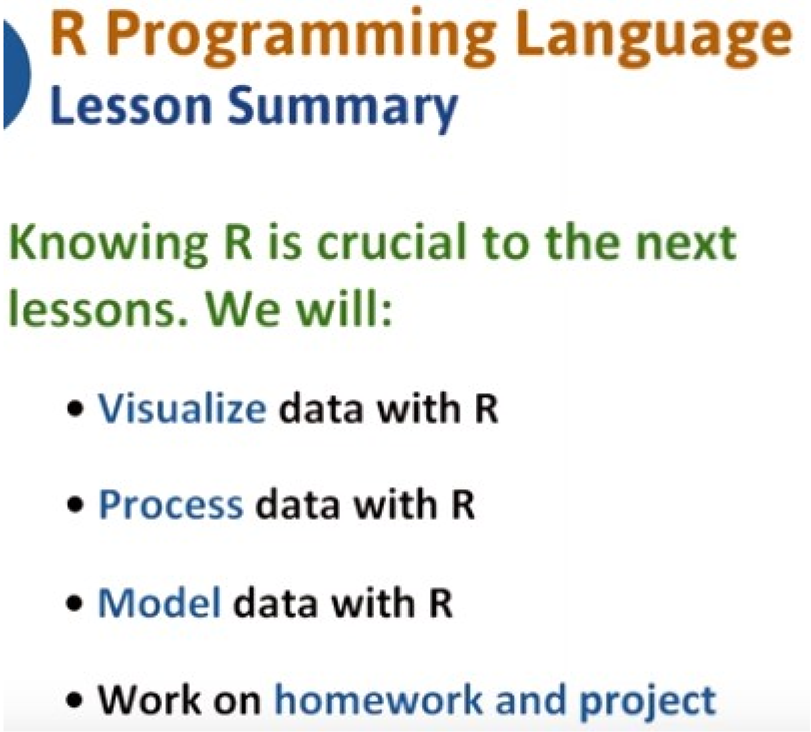
55. Often 10% of the code is responsible for 90% of the computing time. This case is called the bottleneck. The 10% of the code responsible for the 90% of computing time is the bottleneck of the program. If we cannot implement the bottleneck with a vectorized code that accelerates it efficiently, another alternative is to implement the bottleneck C,C++ code and then call that code from within R. let's an example here without some of the details you can find the details in the handouts linked from the courses website. The first line here we load a compiled module having the compiled bottleneck code was written originally in C, compiled by the C compiler into object code. Here we defined two vectors and we call the compiled code with the vectors in R when R sees the function Call() it looks for that module, passes the corresponding vectors, execute it. But notice that the execution here is an execution of compiled code without an interpreter overhead and returning the results. Call() is a relatively old native API. There are some newer alternatives such as Rcpp and there are other packages, RcppArmadillo, RcppEigen that allow you to use some of Rcpp functionality for handling large arrays. Rcpp is slightly more complex than Call unless you use more functionality. In the next slide, we'll see the difference in execution time between the compiled code and running the computational bottleneck in R.



```
## [1] 13.34 17.48 21.21 24.71 28.03 31.24 34.34 37.37 40.33 43
## [1] 13.34 17.48 21.21 24.71 28.03 31.24 34.34 37.37 40.33 43
```

External/ Native API

In the graph here, the x axis corresponds with array size. And the y axis correspond to the computation time in second of a program. We coded that program one time in R and the execution correspond to this dashed line here and one time in C compiled it. And then loaded and executed from within R using the .col interface. The execution of that bottleneck is represented using the solid line that we see here. It's much more computationally efficient and scale up much better. At the top here, we see the output of the two programs. And we see that their identical outputs. The detailed example is available in the handout if you want to look at the code of the R program and the C program. This idea of implementing a bottleneck in C++ and calling it from R is particularly useful, if the bottleneck is very, very very time consuming and relatively simple to implement. We can get the benefit of both worlds, fast code, and at the same time, use a higher level language like R for most of our programming. Except for the bottleneck, which is in many cases a small part.



R Programming Language

Lesson Summary

Knowing R is crucial to the next lessons. We will:

- Visualize data with R
- Process data with R
- Model data with R
- Work on homework and project

56. We saw how to write basic programs in R. We also saw that R can sometimes be very slow and how to handle these cases with calling compiled code that implements computational bottlenecks. Now that you have a basic knowledge of R, we will learn how to use it more specifically to visualize and analyze data. In the process, you will also work on homework and quizzes that will solidify your knowledge. We will next focus on data visualization using R, and then how to process data. After that, we will focus on modeling data where we will also use R to apply the concepts that we will learn in practice. We will also use R in the project and various homework assignments throughout the course.