UCLA QCBio Collaboratory

Introduction to R and data visualization

Wenbin Guo
Bioinformatics IDP, UCLA
wbguo@ucla.edu
2024 Spring

UCLA Bioinformatics

Notation of the slides

- `Code` or `Pseudo-Code` chunk starts with " ➢ ", e.g.
  - `print("Hello world!")`

- Link is underlined

- Important terminology is in **bold** font

- Practice comes with

Workshop goals

- Master the basics of R programming

- Use R data science toolbox for data wrangling

- Present and report results with R data visualization

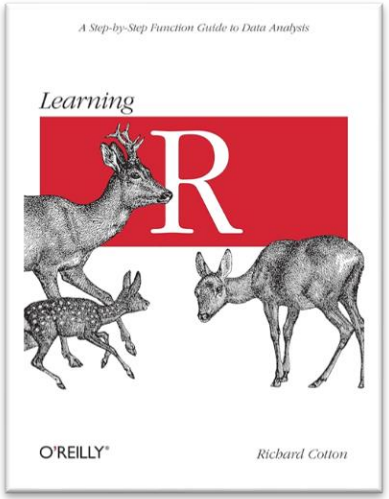
Agenda

- Day 1: R basics
  - Environment setup
  - Variable, Operators
  - Data structure: Vector, Matrix, List, Data frame
- Day 2: R advanced topics
  - Flow control, Loops
  - Function, Packages, File Input/Output
  - Data wrangling with tidyverse toolkit
- Day 3: Data visualization with ggplot2
  - ggplot2 syntax, grammar, and elements
  - Basic plot types and customization

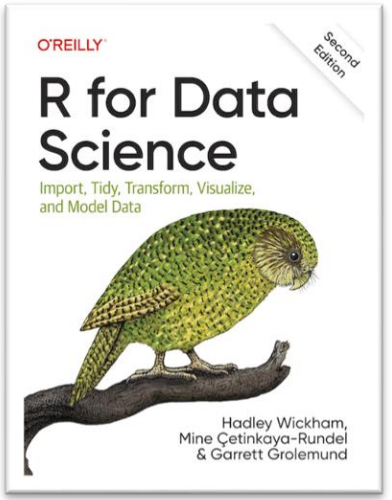
Image from internet

References

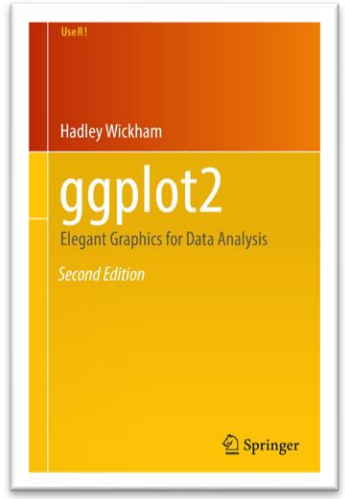
- Main references



link

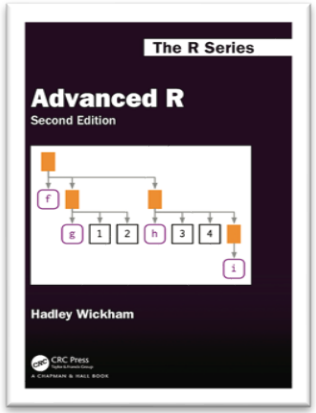


link

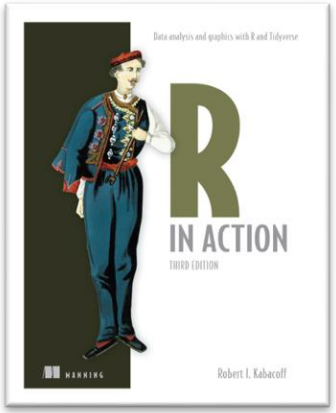


link

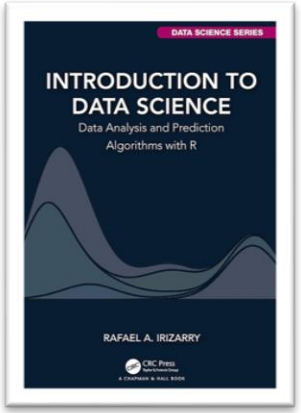
- Other useful references



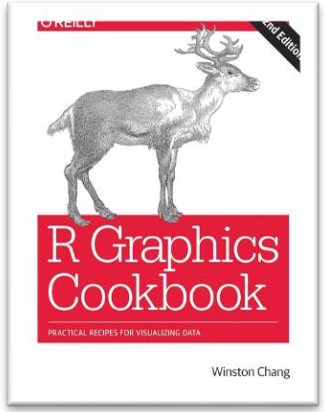
link



link



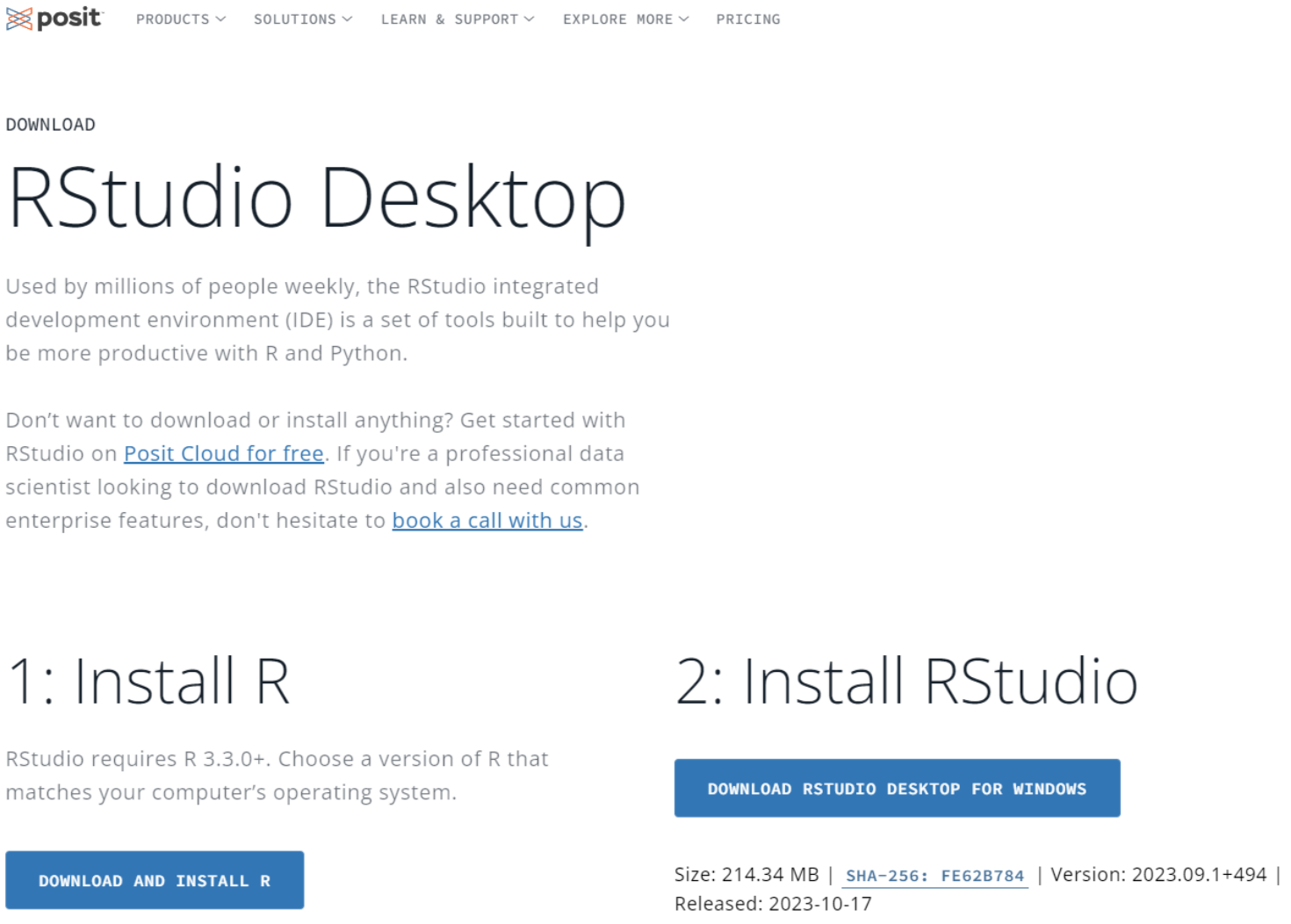
link



link

Environment setup

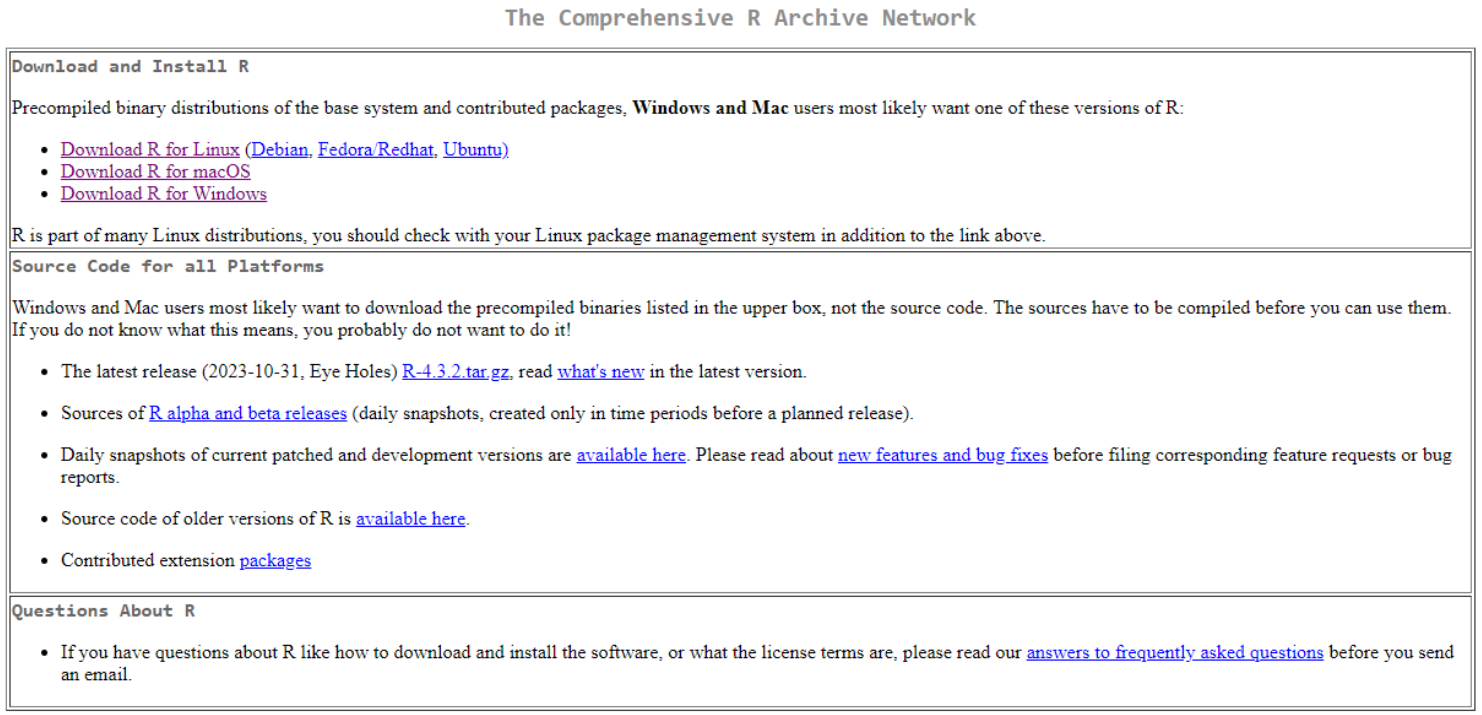
- Go to the official download website



Environment setup

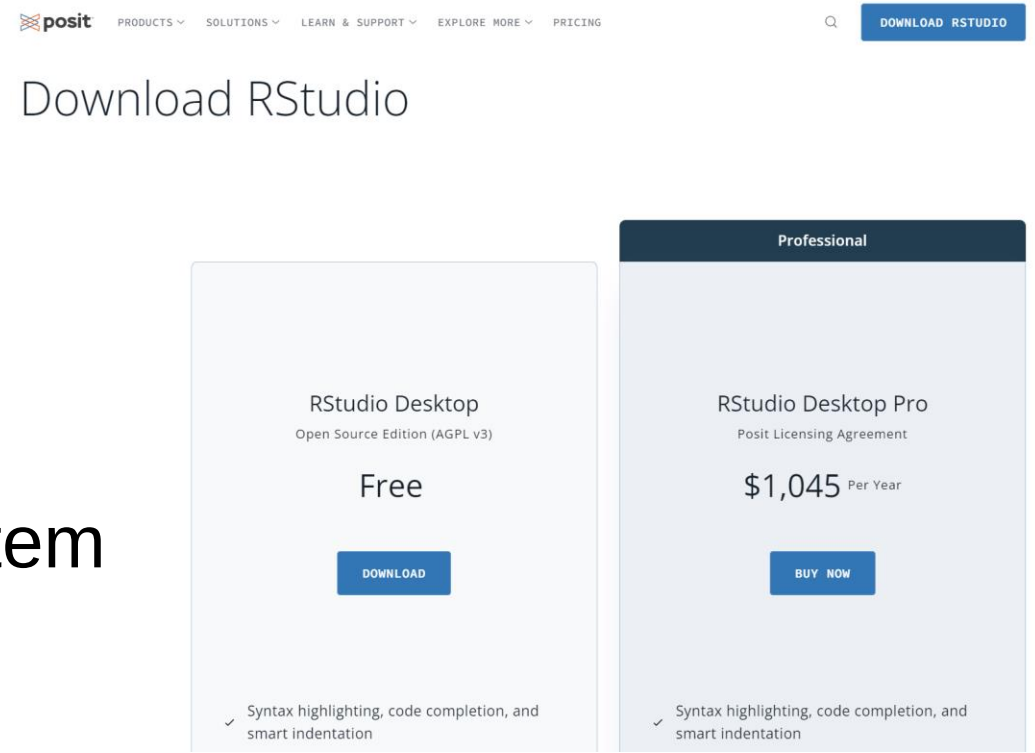
- Go to the official download website

- Install R based on your operating system



Environment setup

- Go to the official download website



- Install R based on your operating system

- Download and install RStudio desktop (Open Source Edition)

UCLA QCBio Collaboratory

Day 1: R basics

Wenbin Guo
Bioinformatics IDP, UCLA
wbguo@ucla.edu
2024 Spring

Overview

Time

- 3-hour workshop (45min + 45min + 30min + practice/Q&A)

Topics

- ❑ Introduction to R and RStudio
  - ❑What's that?
  - ❑Why do we learn?
- ❑ R data structures and operators
  - ❑Variables and Operators
  - ❑Vector, Matrix, List, Data frame
- ❑ Examples and Practices

A brief history of R language

John Chambers

Trevor Hastie

Ross Ihaka

Robert Gentleman

Hadley Wickham

Timeline with selected milestones

1975
The S language is developed at the Bell Labs

1988
"The New S Language" book by John Chambers defines the features of the mature S

1993
Ross Ihaka and Robert Gentleman first mention R, a new implementation of S

1995
R is released under the GNU license

1996
The first official R paper "R: A Language for Data Analysis and Graphics" is published on the Journal of Computational and Graphical Statistics

1997
Founding of CRAN, the R mailing lists and the R core team

2000
R 1.0.0 first stable release

2001
Creation of Bioconductor

2004
R 2.0.0 released, introducing lazy loading and big data support

2007
Hadley Wickham releases ggplot2

2011
Rstudio beta 0.92 is first released

2012
Release of the *shiny* package

2013
R 3.0.0 released with full support for 64bit architectures

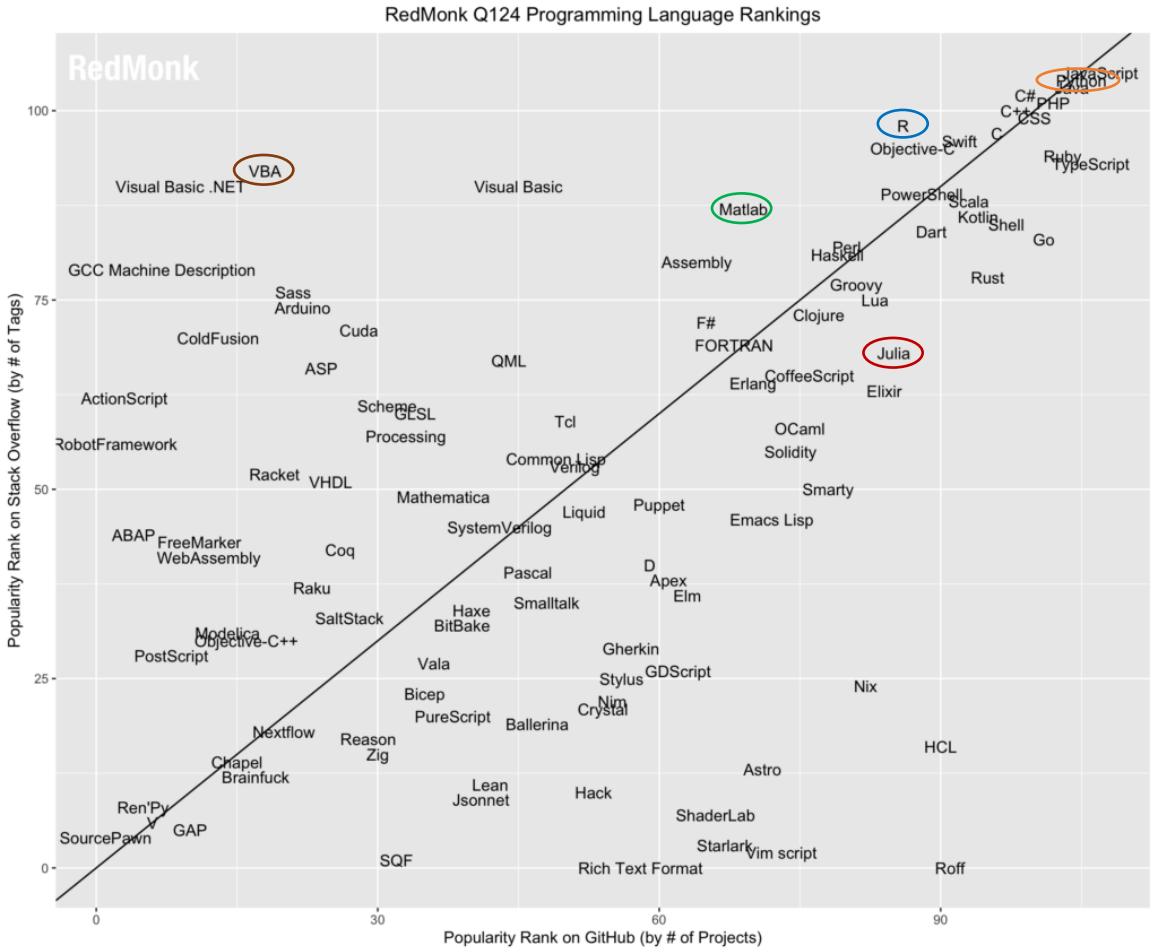
2014
Rmarkdown stable release

2016
Tidyverse package set version 1.0.0

2020
R.4.0.0 is released

Giorgi, et al. *Life* (2022)

R is a popular data science language



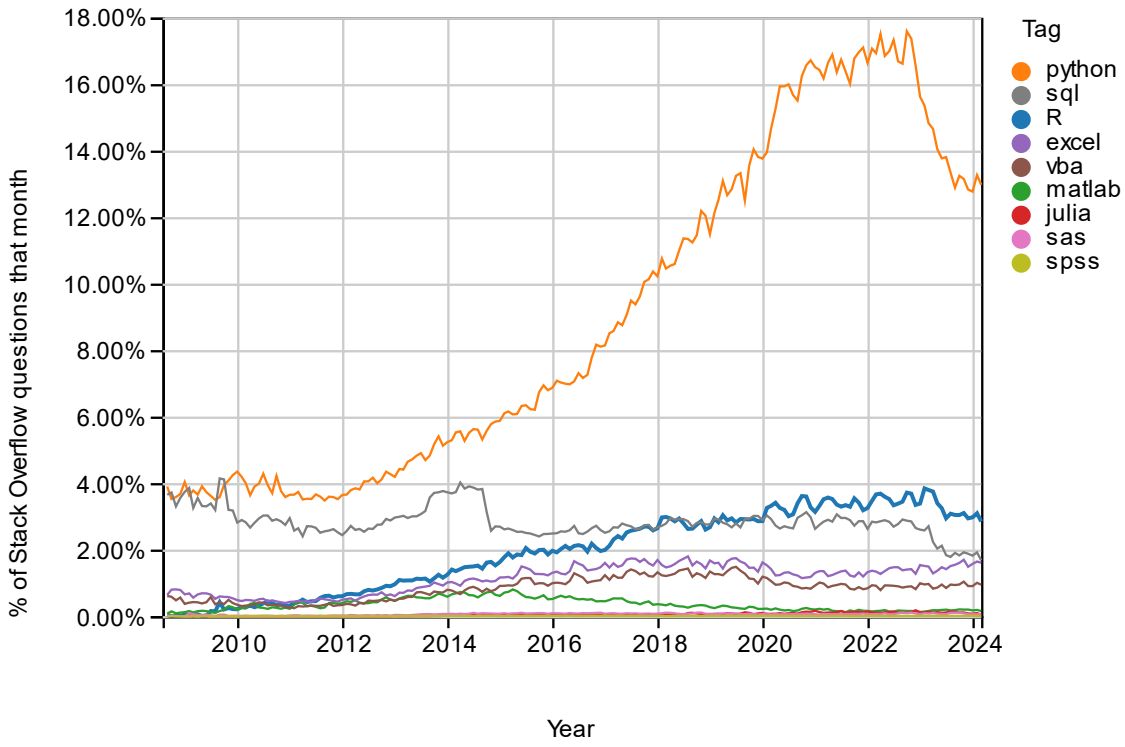
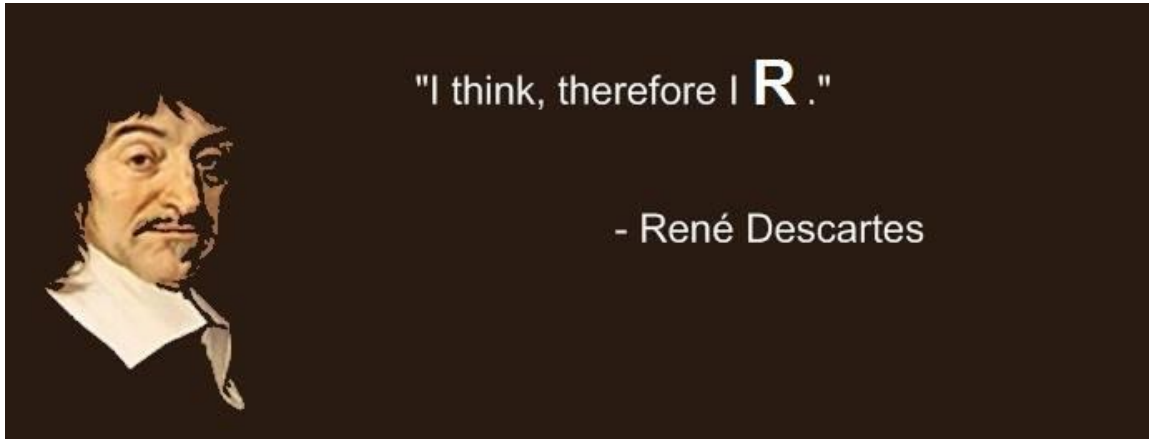


Image from Internet and stackoverflow

Why learning R?

As a wise man once said…

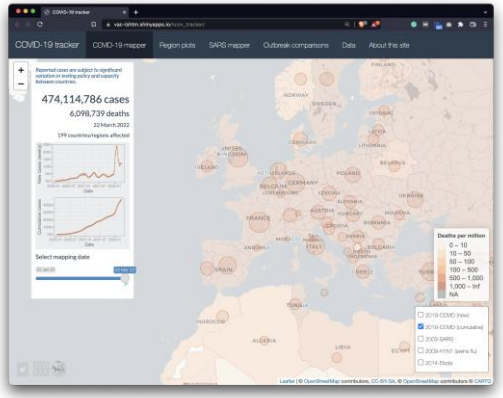


Just kidding…

Why learning R?

- Open-source software
- Well documented tools for analytical tasks
- Rich package ecosystem
- Active and supportive community
- Wide applications in both academia and industry

…



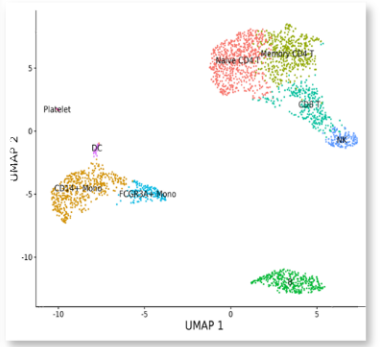
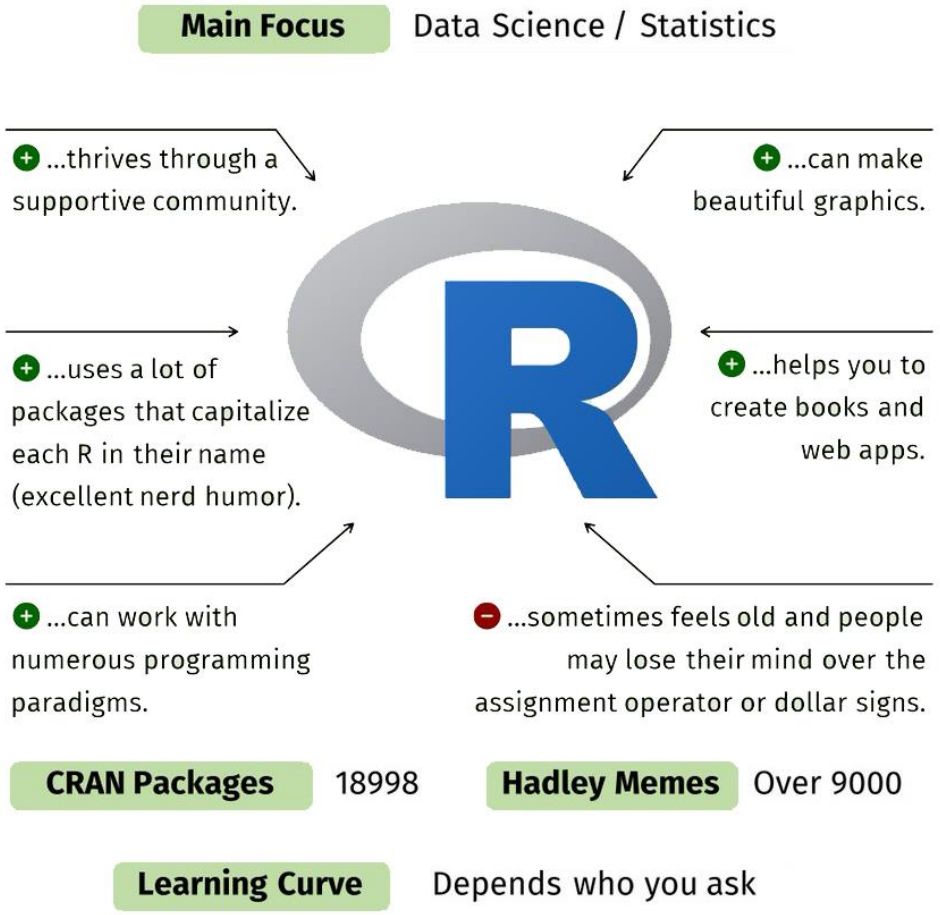


Image from Internet and http://www.r-clinical-research.com/

What you should know about R…



Pros	Cons
Excel at statistical analysis and data visualization	Steep learning curve
Rich package ecosystem	Performance limitations
Supportive community	Memory management limitations
Flexibility and extensibility	Data security concern
Suitable for reproducible research	Limited Object-Oriented Programming support
Cost-effective solutions	Limited GUI options
Active development	Debugging challenges
…	…

Image from internet

R vs Python?

	R	Python
Pros	• Best for data visualization • Easy to do with packages • Active community • Extended applications within statistics and data science • More people work alike: using the same IDE and the same workflow with the Tidyverse	• Best for experienced coders to pick up • Best for machine learning • Best for beginners who want to explore the world of programming • Larger global user base • Easier to collab with other programmers in the team • Best to deploy algorithms
Cons	• Poorly written code affects speed • Can be time-consuming to develop expertise • Can be hard to pick the most appropriate package when more than one exists that accomplishes the same task • Harder to understand and contribute to the work of colleagues using another programming language	• Less appealing data visualisations • Less specialised packages for statistical analysis compared to R • Harder to know which IDE to use and which libraries to pick

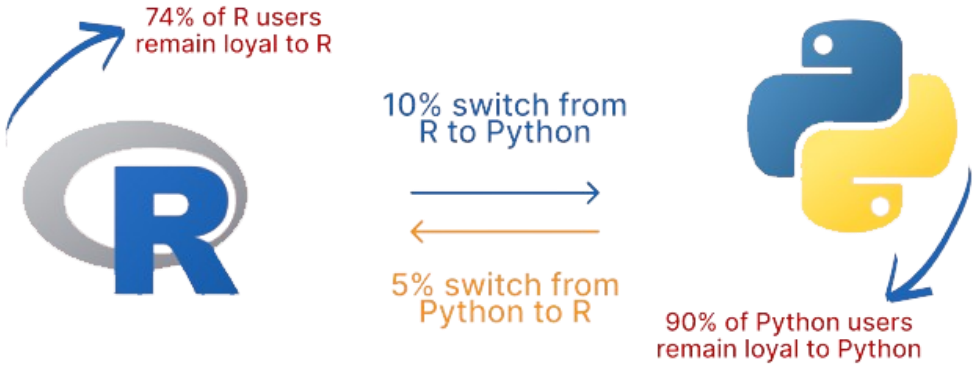


Image from Internet

R vs Python?

	R	Python
Pros	• Best for data visualization • Easy to do with packages • Active community • Extended applications within statistics and data science • More people work alike: using the same IDE and the same workflow with the Tidyverse	• Best for experienced coders to pick up • Best for machine learning • Best for beginners who want to explore the world of programming • Larger global user base • Easier to collab with other programmers in the team • Best to deploy algorithms
Cons	• Poorly written code affects speed • Can be time-consuming to develop expertise • Can be hard to pick the most appropriate package when more than one exists that accomplishes the same task • Harder to understand and contribute to the work of colleagues using another programming language	• Less appealing data visualisations • Less specialised packages for statistical analysis compared to R • Harder to know which IDE to use and which libraries to pick

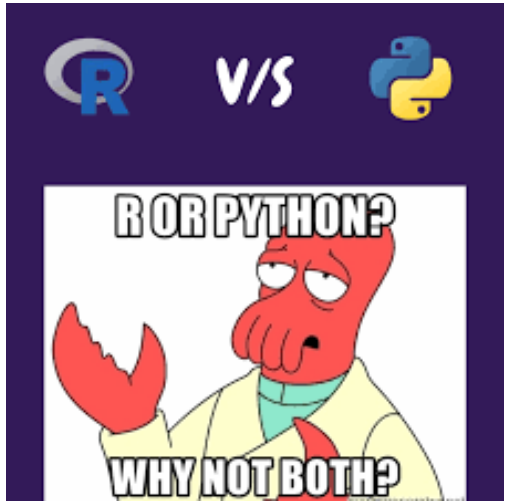
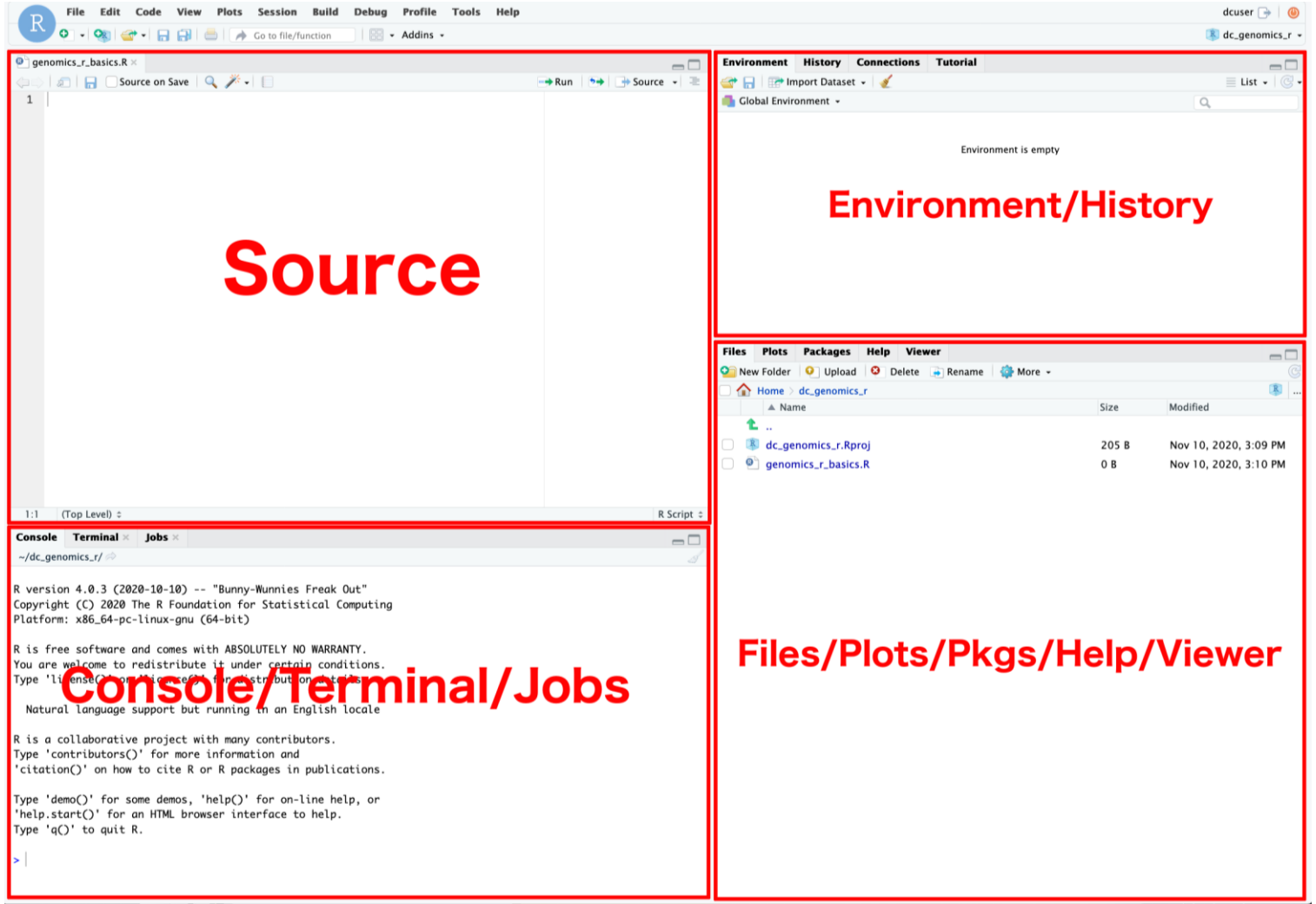


Image from Internet

RStudio IDE

- IDE: Integrated Development Environment



Your first R program

>hello world.■

- Open `R/RStudio` interface

- Say “Hello world!”
  - `print("Hello world!")`

- Exit
  - `q()`

Get help in R

- `?fun_name`: if you know the function/dataset name
- `??keyword`: if you don't know the name, but know the topic
- `help()/help.search()`: similar to `?` and `??` symbol, respectively
- `apropos("keyword")`: if you half remember the variable/function's name

```
?mean                   help("mean")                    #opens the help page for the mean function
?"+"                    help("+")                       #opens the help page for addition
?"if"                   help("if")                      #opens the help page for if, used for branching code
??plotting              help.search("plotting")         #searches for topics containing words like "plotting"
??"regression model"    help.search("regression model") #searches for topics containing phrases like this
```

```
apropos("vector")

## [1] ".__C__vector"          "a_vector"              "as.data.frame.vector"
## [4] "as.vector"             "as.vector.factor"      "is.vector"
## [7] "vector"                "Vectorize"
```

Comments

The hashtag / pound (#) symbol starts the comment, R will ignore the rest of the line

- Useful for documentation and communication

➢ `print("Hello R!") # will print “Hello R!”`

Question: what if you have multiple-lines to comment?

Shortcut: select the lines and press

- Windows: `Ctrl + Shift + C`
- MacOS: `Cmd + Shift + C`

Variable

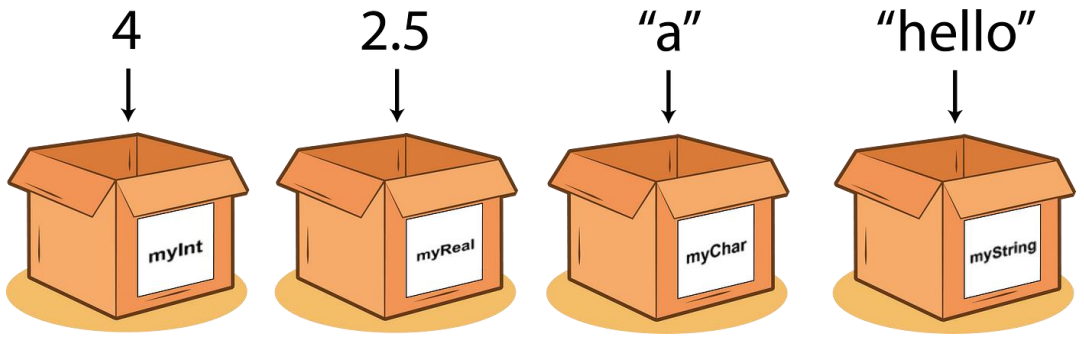


Image from Internet

Variable name

TwoHardThings

Martin Fowler
14 July 2009

> *There are only two hard things in Computer Science: cache invalidation and naming things.*
>
> *-- Phil Karlton*

Naming rules

- Variable names can contain letters, numbers, dots, and underscores
- But they can't start with a number, or a dot followed by a number (since that looks too much like a number)
- Reserved words like "if " and "for" are not allowed

When you try to choose a meaningful variable name.

Check details about what is and isn't allowed by

➢ `?make.names`

Tips:

- name meaningfully
- `make.names` is useful with duplicate names

Value assignment

- We can assign a (local) variable using either `<-` or `=`, though for historical reasons, `<-` is preferred
  - Normally, they can be used interchangeably
  - Note there is no space between `<` and `-`
  - Some surrounding space would be nice

```
x <- 3
x < -3
x<-3     #is this assignment or less than?
```

- We can assign a (global) variable using `<<-`

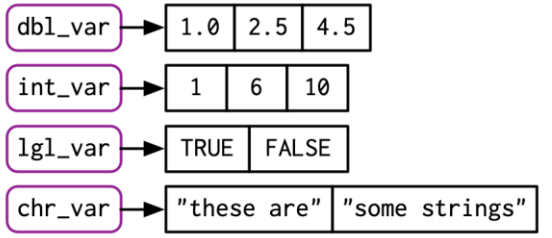
Operator	Coverage	Description
`<-`	local/global	the value from the right hand side is inserted into the object on the left hand side
`->`	local/global	the value from the left hand side is inserted into the object on the right hand side
`<<-`	global	the value from the right hand side is inserted into the global object on the left hand side
`->>`	global	the value from the left hand side is inserted into the global object on the right hand side

- We can also assign variable values via the `assign()` function
  - `assign("mylocal_var", 9); assign("myglobal_var", 99, globalenv())`

Let’s do some practice!

- `git clone https://github.com/wbvguo/qcbio-Intro2R.git`

Variable classes



All variables in R have a class, indicating their variable types

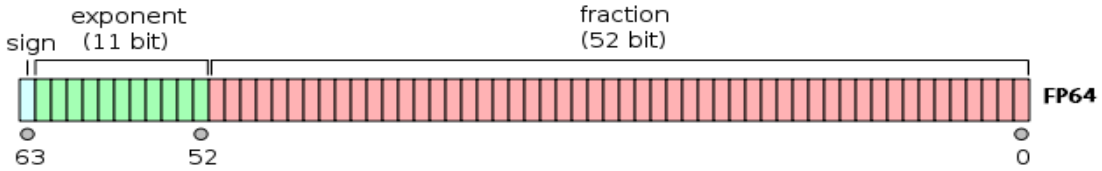
- **Logical**: storing TRUE/FALSE
- **Numbers**
  - Numeric: floating point values
  - Integer: integer (add 'L' suffix to make numbers as integers)
  - Complex: complex numbers (add 'i' to create imaginary components)
- **Character**: storing text
- **Factor**: storing integer with labels (more memory efficient)
- …

Use `class()` function to check classes

```
class(sqrt(1:10))
## [1] "numeric"
class(3 + 1i)      #"i" creates imaginary components of complex numbers
## [1] "complex"
class(1)           #although this is a whole number, it has class numeric
## [1] "numeric"
class(1L)          #add a suffix of "L" to make the number an integer
## [1] "integer"
class(0.5:4.5)     #the colon operator returns a value that is numeric...
## [1] "numeric"
class(1:5)         #unless all its values are whole numbers
## [1] "integer"
```

Numerical value's representation in machine

A double precision (64 bit) number in machine is represented as



$$\pm d_0 . d_1 d_2 \cdots d_p \times b^e$$

- The biggest number a double precision number can represent is around 1.8e308
- The smallest positive number it can represent is 2.2e-308

Check the properties of R's numbers

- `.Machine`

Issues?

Usually it's enough for daily usage, but sometimes the rounding error needs to be considered

Special numbers

- `Inf`: positive infinity
- `-Inf`: negative infinity
- `NA`: Not available (missing value)
- `NaN`: Not a number (the calculation doesn't make sense mathematically)

Try it out:

- Type in 1/0 in Console, what would you get?
- What happens if we add, subtract, multiply, divide a NA?

Other interesting numbers

- `pi`: circumference to diameter ratio (~3.141593)
- Question: how to get the natural exponent `e`?
- `i`: imaginary number `(1i)`

Inspect variables

- Inspect the class of variables
  - `class(my_var)`
  - `is.numeric(); is.character(); is.logical()… #is.*`

- Inspect the values
  - `print(my_var)`
  - `head(my_car) # only print the first several values`

- Inspect the structure
  - `str(my_var)`

Inspect variables

- Inspect the size (memory allocation)
  - `object.size(my_var)`

```
gender_char <- sample(c("female", "male"), 10000, replace = TRUE)
gender_fac <- as.factor(gender_char)
object.size(gender_char)

## 80136 bytes

object.size(gender_fac)

## 40512 bytes
```

- Convert between types (`as.*` function)
  - `as.numeric()`
  - `as.integer()`
  - `as.bool()`
  - `as.character()`
  - …

```
x <- "123.456"
as(x, "numeric")

## [1] 123.5

as.numeric(x)

## [1] 123.5
```

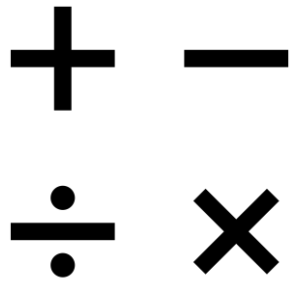
Inspect workspace

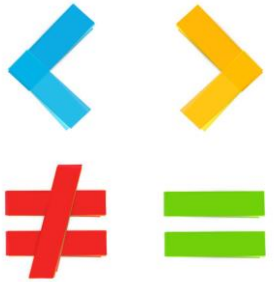
- Get the work directory
  - `getwd()`
- Set the work directory
  - `setwd("path/to/the/directory")`
- List the names of existing variables
  - `ls()`
- Remove a variable
  - `rm(var_name)`

Question: How to clean the environment?

- `rm(list = ls())`

Operators





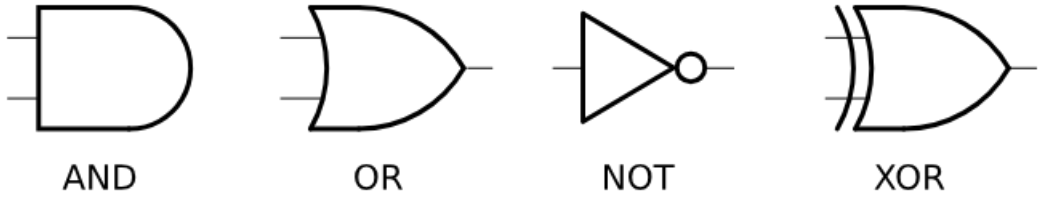


Image from Internet

Arithmetic operators

- For vectors (a scalar variable is regarded as a vector of length 1)
  - Element-wise operations

```
c(2, 3, 5, 7, 11, 13) - 2          #subtraction
## [1]  0  1  3  5  9 11
```

Operator	Description
+	addition
-	subtraction
*	multiplication
/	division
^ or **	exponentiation
x %% y	modulus (x mod y) 5%%2 is 1
x %/% y	integer division 5%/%2 is 2

- For matrix
  - Element-wise operation: `*`
  - Element-wise inverse: `^-1`
  - Inner Matrix multiplication: %*%
  - Outer Matrix multiplication: %o%
  - Matrix inverse: `solve(mat)`

Try it out:

- Verify Euler's equation $e^{i\pi}+1=0$

Relational operators

Operator	Description
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	exactly equal to
!=	not equal to

- Comparison
  - Comparing non-integers

```
sqrt(2) ^ 2 == 2          #sqrt is the square-root function
```

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

  - What happened?

```
sqrt(2) ^ 2 - 2          #this small value is the rounding error
```

```
## [1] 4.441e-16
```

  - Solution: R also provides the function `all.equal()` for checking equality of numbers, ignore the difference if it's small than tolerance level (1.5e-8, by default)

```
all.equal(sqrt(2) ^ 2, 2)
```

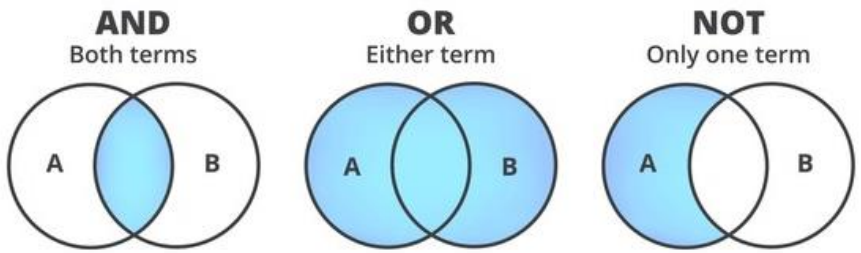
```
## [1] TRUE
```

- Membership
  - Belong to: `%in%`

Question: what about not belong to?

Logical operators

- Simple
  - !: NOT
  - &: AND
  - |: OR



Operator	Description
!x	Not x
x | y	x OR y
x & y	x AND y
isTRUE(x)	test if X is TRUE

- Compound
  - `any()`: `TRUE` if at least one of the values in input is `TRUE`
  - `all()`: `TRUE` if all values in input is `TRUE`

Operator's precedence

Question: what will the following code output?

- 1 + 2 * 3
- 1==2 & 1

Note:

You can use ( ) to encapsulate the part you want to compute first

- (1+2)*3
- 1==(2 &1)

Precedence	Operator	Description
18	:: :::	access variables in a namespace
17	$ @	component / slot extraction
16	[ ] [[ ]]	indexing
15	^	Exponentiation operator (Right to Left)
14	+a -a	Unary plus, Unary minus
13	:	Sequence operator
12	%% %*% %/% %in% %o% %x%	Special operators
11	* /	Multiplication, Division
10	+ -	Addition, Subtraction
9	< <= > >=	Less than, Less than or equal, Greater than, and Greater than or equal
	== !=	Equality and Inequality
8	!	Logical NOT
7	& &&	Logical AND
6	| ||	Logical OR
5	~	as in formulae
4	-> ->>	Right assignment operator, Global right assignment operator
3	<- <<-	Left assignment operator, Global left assignment operator (Right to Left)
2	=	Left assignment operator (Right to Left)
1	?	help (unary and binary)

Top to bottom in **descending precedence**

Image from Internet

R Objects

Data Structures

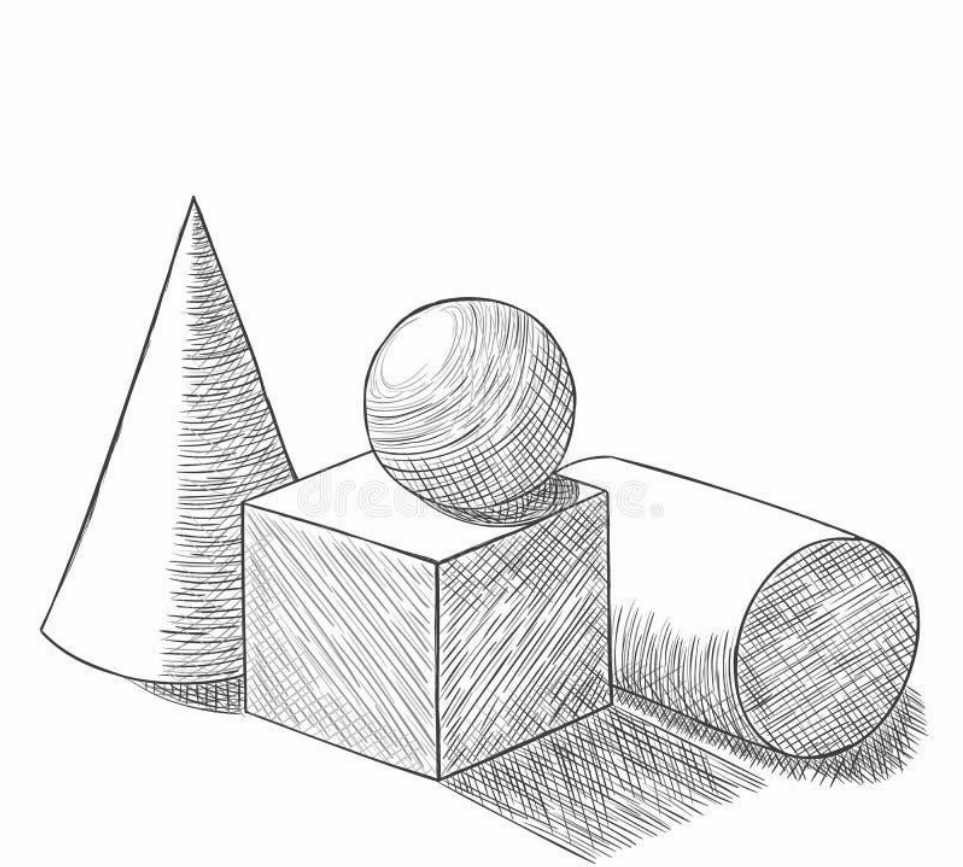
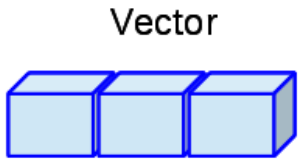


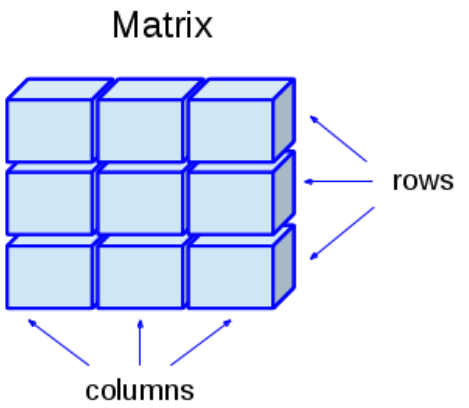
Image from Internet

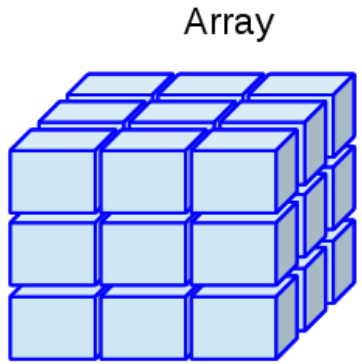
Objects in R

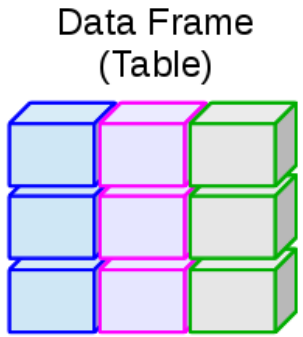
Actions

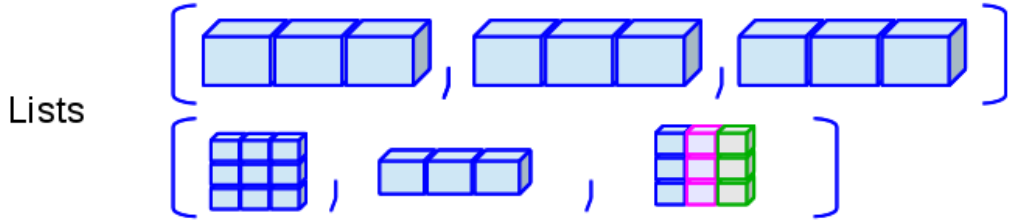
- Create
- Indexing
- Update
- …









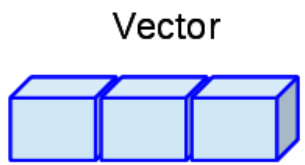


http://venus.ifca.unican.es/Rintro/dataStruct.html

Vector - create

- Create an "empty" vector
  - `vector(type, length)`

  Each values in the result is zero, FALSE, or an empty string, or the equivalent of “nothing”



1d data structure

```
vector("numeric", 5)

## [1] 0 0 0 0 0

vector("complex", 5)

## [1] 0+0i 0+0i 0+0i 0+0i 0+0i

vector("logical", 5)

## [1] FALSE FALSE FALSE FALSE FALSE

vector("character", 5)

## [1] "" "" "" "" ""

vector("list", 5)

## [[1]]
## NULL
##
## [[2]]
## NULL
##
## [[3]]
## NULL
##
## [[4]]
## NULL
##
## [[5]]
## NULL
```

Vector - create

- Create an "empty" vector
  - `vector(type, length)`

    Each values in the result is zero, FALSE, or an empty string, or the equivalent of “nothing”

- Create a sequence vector
  - `1:10`
  - `seq(from, to, by, length.out)`

- Manually create a vector
  - `c(1, 2, 3)`

- Create a repetition vector
  - `rep(…)`

```
8.5:4.5                 #sequence of numbers from 8.5 down to 4.5
## [1] 8.5 7.5 6.5 5.5 4.5
c(1, 1:3, c(5, 8), 13) #values concatenated into single vector
## [1]  1  1  2  3  5  8 13
```

```
rep(1:5, 3)
##  [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
rep(1:5, each = 3)
##  [1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
rep(1:5, times = 1:5)
##  [1] 1 2 2 3 3 3 4 4 4 4 5 5 5 5 5
rep(1:5, length.out = 7)
## [1] 1 2 3 4 5 1 2
```

Vector - length

- Access the number of elements in the vector
  - `length()`

  Missing values still count toward the length

```
sn <- c("Sheena", "leads", "Sheila", "needs")
length(sn)

## [1] 4

nchar(sn)

## [1] 6 5 6 5
```

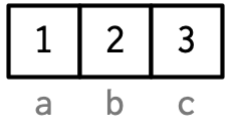
- Assign new length to existing vector
  - Shorten a vector: values at the end will be removed
  - Extend a vector: missing values will be added to the end

```
poincare <- c(1, 0, 0, 0, 2, 0, 2, 0)
length(poincare) <- 3
poincare

## [1] 1 0 0

length(poincare) <- 8
poincare

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



Vector - name

- Assign names when creating a vector

```
c(apple = 1, banana = 2, "kiwi fruit" = 3, 4)
```

```
##      apple     banana kiwi fruit
##          1          2          3          4
```

- Assign names to existing vector

```
x <- 1:4
names(x) <- c("apple", "bananas", "kiwi fruit", "")
x
```

```
##      apple    bananas kiwi fruit
##          1          2          3          4
```

- Retrieve names from a vector

```
names(x)
```

```
## [1] "apple"      "bananas"    "kiwi fruit" ""
```

Vector - index/subset/slice

Accessing part of a vector by []. One can pass

- a vector of **positive numbers** returns the slice of the vector containing the elements at those locations. (R indices start with 1)
- a vector of **negative numbers** returns the slice of the vector containing the elements everywhere *except* at those locations
- **a logical vector** returns the slice of the vector containing the elements where the index is TRUE
- **a character vector** of names returns the slice of the named vector containing the elements with those names (like python dict)

```
x <- (1:5) ^ 2
## [1]  1  4  9 16 25
```

```
x[c(1, 3, 5)]
x[c(-2, -4)]
x[c(TRUE, FALSE, TRUE, FALSE, TRUE)]
## [1]  1  9 25
```

```
names(x) <- c("one", "four", "nine", "sixteen", "twenty five")
x[c("one", "nine", "twenty five")]
##          one         nine twenty five
##            1            9           25
```

Vector - index/subset/slice

Note:

- Mixing positive and negative values is not allowed
- Missing indices correspond to missing values in the result
- Out of range indices, beyond the length of the vector, don’t cause an error, but instead return the missing value NA
- Non-integer indices are silently rounded toward zero

Useful functions:

- `which()`: returns the locations where a logical vector is TRUE
- `which.min(), which.max()`: returns the locations where the min and max value located

Vector - recycle

What would happen if you run the following code?

- `1:5 + 1`
- `1:5 + 5:1`
- `1:5 + 1:10`
- `1:5 + 1:6`

- When adding two vectors together, R will recycle elements in the shorter vector to match the longer one

- If the length of the longer vector isn't a multiple of the length of the shorter one, a warning will be given

Vector - combine

`c()` stands for concatenate, a Latin word meaning “connect together in a chain”

Use `c()` to concatenate 2 vectors into 1 vector

- `c(c(1,2,3), 4)`
- `c(c(1,2,3), c(4,5,6))`

String and factor

Character vector: storing text `c("red", "yellow")`

Useful function

- The `paste()` function combines strings together, with a separator, and a collapse
- The `paste0()` function combines strings without separator
- The `toupper()` and `tolower()` function change cases
- The `strsplit()` function split string into a list based on supplied patterns

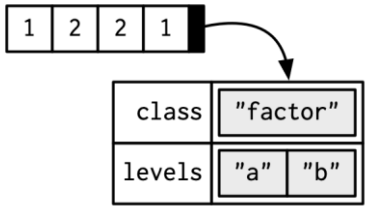
```
paste(c("red", "yellow"), "lorry")
## [1] "red lorry"    "yellow lorry"
paste(c("red", "yellow"), "lorry", sep = "-")
## [1] "red-lorry"    "yellow-lorry"
paste(c("red", "yellow"), "lorry", collapse = ", ")
## [1] "red lorry, yellow lorry"
paste0(c("red", "yellow"), "lorry")
## [1] "redlorry"    "yellowlorry"
```

```
toupper("I'm Shouting")
## [1] "I'M SHOUTING"
tolower("I'm Whispering")
## [1] "i'm whispering"
```

String and factor

Factor: storing categorical data, storing integer with labels

- Create a factor using `factor()` function
- Access levels using `level()` function




```
gender_char <- c(
  "female", "male", "female", "male", "male",
  "female", "female", "male", "male", "female"
)
(gender_fac <- factor(gender_char))
```

```
##  [1] female male   female male   male   female female male   male   female
## Levels: female male
```

By default, factor levels are assigned alphabetically, to change the order of factor levels, specify the levels argument

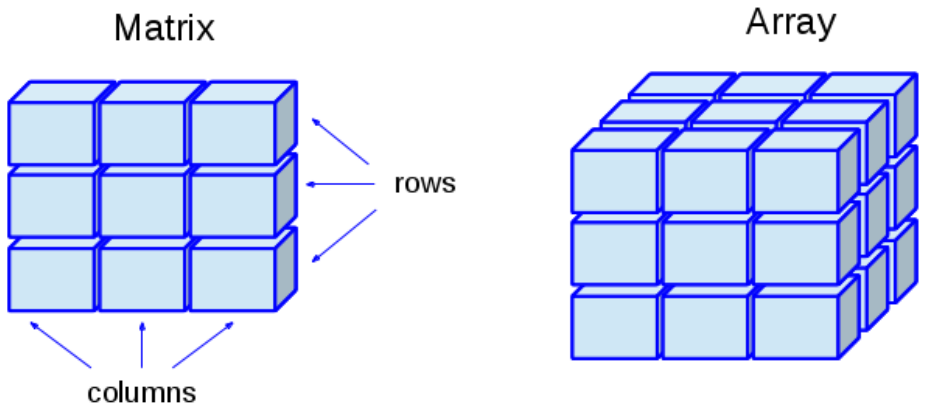
```
factor(gender_char, levels = c("male", "female"))
```

```
##  [1] female male   female male   male   female female male   male   female
## Levels: male female
```

Array and Matrix

Arrays hold multidimensional rectangular data. Matrix is a special case of Arrays (2-dimensional)

- “Rectangular” means that each row is the same length, and likewise for each column and other dimensions
- Like vectors, matrix holds values of the same data type



Matrix - create

- `matrix(vectors, nrow, ncol, dimnames)`

Note:

- When creating a matrix, the values fill the matrix column-wise. To fill the matrix row-wise, one can specify the argument by`row = TRUE`

```
(a_matrix <- matrix(
  1:12,
  nrow = 4,            #ncol = 3 works the same
  dimnames = list(
    c("one", "two", "three", "four"),
    c("ein", "zwei", "drei")
  )
))
```

```
##       ein zwei drei
## one     1    5    9
## two     2    6   10
## three   3    7   11
## four    4    8   12
```

```
class(a_matrix)
```

```
## [1] "matrix"
```

Matrix - dimensions and names

Useful function:

- Access the dimension
  - `dim(mat)`
  - `nrow(mat)`
  - `ncol(mat)`

- Reshape matrix
  - `dim(mat) =c(new_dim1, new_dim2)`

- Access the row/column names
  - `rownames()`
  - `colnames()`
  - `dimnames()`

```
dim(a_matrix)

## [1] 4 3

nrow(a_matrix)

## [1] 4

ncol(a_matrix)

## [1] 3
```

```
dim(a_matrix) <- c(6, 2)
a_matrix

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

```
rownames(a_matrix)

## [1] "one"   "two"   "three" "four"

colnames(a_matrix)

## [1] "ein"  "zwei" "drei"

dimnames(a_matrix)

## [[1]]
## [1] "one"   "two"   "three" "four"
##
## [[2]]
## [1] "ein"  "zwei" "drei"
```

Matrix - index and combine

- Similar to vectors, we can use square brackets [] to access part of matrix, and we still have four index choices (positive integers, negative integers, logical values, element names)

```
a_matrix[1, c("zwei", "drei")] #elements in 1st row, 2nd and 3rd columns

## zwei drei
##    5    9
```

- To include all of a dimension, leave the corresponding index blank

```
a_matrix[1, ]                 #all of the first row

##  ein zwei drei
##    1    5    9

a_matrix[, c("zwei", "drei")]  #all of the second and third columns

##       zwei drei
## one      5    9
## two      6   10
## three    7   11
## four     8   12
```

- Combine matrix
  - `cbind(mat1, mat2) # bind by column`
  - `rbind(mat1, mat3) # bind by row`

List and Data frame

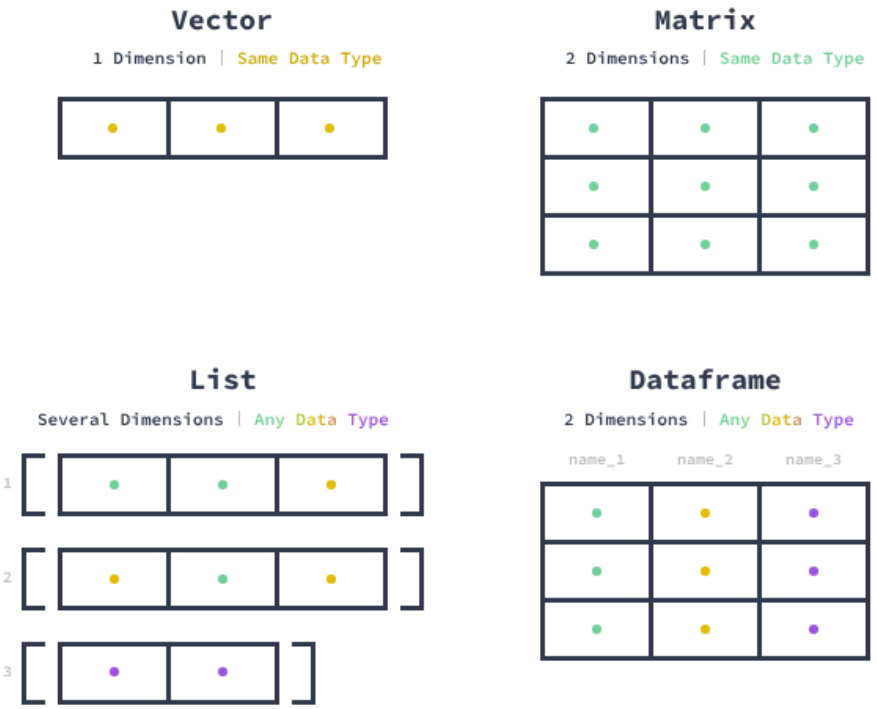
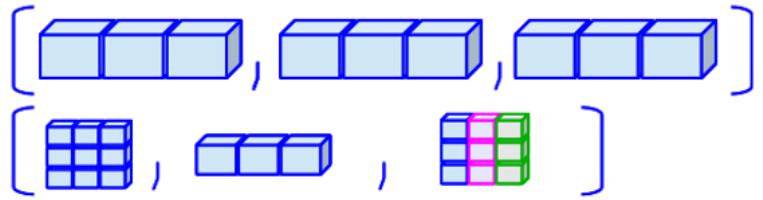


Image from Internet

List - create

A list is, loosely speaking, a vector where each element can be of a different type

Lists



- Create an empty list with `vector()` function
  - `a_list = vector("list", length)`
- Create list with `list()` function, with each argument separated by a comma
  - `a_list = list(a, b, c, d)`

```
(a_list <- list(
  c(1, 1, 2, 5, 14, 42),    #See http://oeis.org/A000108
  month.abb,
  matrix(c(3, -8, 1, -3), nrow = 2),
  asin
))
```

```
## [[1]]
## [1]  1  1  2  5 14 42
##
## [[2]]
##  [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov"
## [12] "Dec"
##
## [[3]]
##      [,1] [,2]
## [1,]    3    1
## [2,]   -8   -3
##
## [[4]]
## function (x)  .Primitive("asin")
```

List - name

- Name elements when creating the list

```
(the_same_list <- list(
  catalan    = c(1, 1, 2, 5, 14, 42),
  months     = month.abb,
  involutary = matrix(c(3, -8, 1, -3), nrow = 2),
  arcsin     = asin
))
```

- Name elements to existing list

```
names(a_list) <- c("catalan", "months", "involutary", "arcsin")
a_list
```

```
## $catalan
## [1]  1  1  2  5 14 42
##
## $months
##  [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov"
## [12] "Dec"
##
## $involutary
##      [,1] [,2]
## [1,]    3    1
## [2,]   -8   -3
##
## $arcsin
## function (x)  .Primitive("asin")
```

List - dimensions and operations

- Like vectors, lists have length, which is the number of top-level elements, and can be accessed by `length()`

```
(main_list <- list(
  middle_list          = list(
    element_in_middle_list = diag(3),
    inner_list             = list(
      element_in_inner_list         = pi ^ 1:4,
      another_element_in_inner_list = "a"
    )
  ),
  element_in_main_list = log10(1:10)
))
```

```
length(main_list) #doesn't include the lengths of nested lists

## [1] 2
```

- Like vectors, lists don't have dimensions. The `dim()` function returns NULL

- Unlike with vectors, arithmetic doesn't work on lists

List - index

- We can access elements of the list using square brackets []. The result of these indexing operations is another list.
- If we want to access the contents of the list elements, use double square brackets ([[]]) with an index (positive number) or a name (string) to the element
- For named elements of lists, we can also use the dollar sign operator $

```
l <- list(
  first  = 1,
  second = 2,
  third  = list(
    alpha = 3.1,
    beta  = 3.2
  )
)
```

```
l[1:2]

## $first
## [1] 1
##
## $second
## [1] 2
```

```
l[[1]]

## [1] 1

l[["first"]]

## [1] 1
```

```
l$first

## [1] 1

l$f     #partial matching interprets "f" as "first"

## [1] 1
```

List - combine list/remove element

The `c()` function that we have used for concatenating vectors also works for concatenating lists

```
c(list(a = 1, b = 2), list(3))

## $a
## [1] 1
##
## $b
## [1] 2
##
## [[3]]
## [1] 3
```

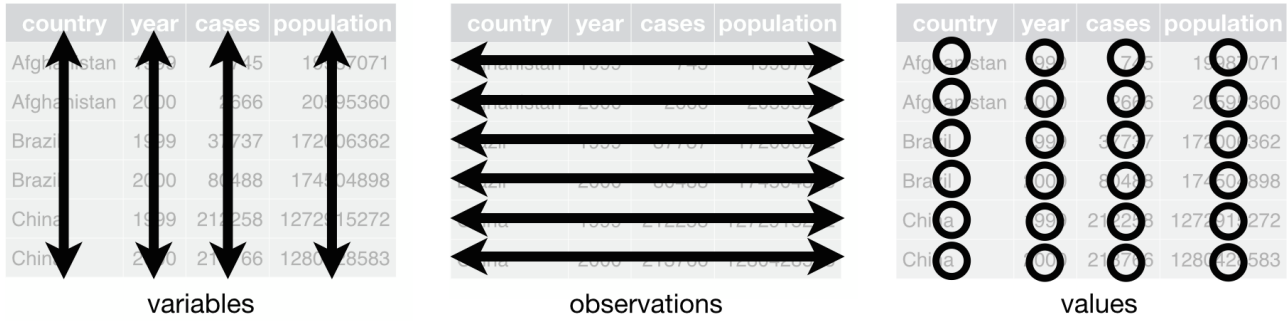
Setting an element to NULL will remove it

Question: difference between NULL and NA?

Data frame - create

Data frames are used to store spreadsheet-like data. They can be thought as

- Matrices where each column can store a different type of data
- Non-nested lists where each element is of the same length



Use function `data.frame()` to create a data frame

- `df = data.frame(x = c(1,2,3), y = c("low", "medium", "high"))`

Note: all the elements within a column are the same type

Image from Internet

Data frame - name

Rows

- When creating the data frame, pass a vector to `row.names` in `data.frame()`
- For existing data frame, get and set the row names using `rownames()`

```
data.frame(
  x = letters[1:5],
  y = y,
  z = runif(5) > 0.5,
  row.names = c("Jackie", "Tito", "Jermaine", "Marlon", "Michael")
)
```

```
##          x       y     z
## Jackie   a -0.9373  TRUE
## Tito     b  0.7314 FALSE
## Jermaine c -0.3030  TRUE
## Marlon   d -1.3307 FALSE
## Michael  e -0.6857 FALSE
```

```
rownames(a_data_frame)
```

```
## [1] "1" "2" "3" "4" "5"
```

```
colnames(a_data_frame)
```

```
## [1] "x" "y" "z"
```

```
dimnames(a_data_frame)
```

```
## [[1]]
## [1] "1" "2" "3" "4" "5"
##
## [[2]]
## [1] "x" "y" "z"
```

Columns

- When creating data frames, column names are checked to be unique. This feature can be turned off by passing `check.names = FALSE` in `data.frame()`
- For existing data frame, get and set the column names using `colnames()`

Get both column and row names using `dimnames()`

Data frame - dimensions

Similar to matrix, the following functions works to get the dimensionality

- `nrow()`: number of rows/observations
- `ncol()`: number of columns/variables
- `dim()`: dimensionality of the data frame

Data frame - index

Similar to matrix indexing

- the four different vector indices (positive integers, negative integers, logical values, and characters) can be used

```
a_data_frame[2:3, -3]
```

```
##   x       y
## 2 b 0.06894
## 3 c 0.74217
```

```
a_data_frame[c(FALSE, TRUE, TRUE, FALSE, FALSE), c("x", "y")]
```

```
##   x       y
## 2 b 0.06894
## 3 c 0.74217
```

Similar to list indexing

- If we only want to select one column, then list-style indexing (double square brackets [[]] with a positive integer or name, or the dollar sign operator $ with a name) can be used

```
a_data_frame$x[2:3]
```

```
## [1] b c
## Levels: a b c d e
```

```
a_data_frame[[1]][2:3]
```

```
## [1] b c
## Levels: a b c d e
```

```
a_data_frame[["x"]][2:3]
```

```
## [1] b c
## Levels: a b c d e
```

If only one column is selected, the result will be simplified to be a vector

Data frame - subset

- Subset by variable conditions (similar to pass a logical values)

- Use `subset()` function, which takes 3 arguments
  - a data frame to subset
  - a logical vector of conditions for rows to include
  - a vector of column names to keep (if omitted, all columns are kept)

```
a_data_frame[a_data_frame$y > 0 | a_data_frame$z, "x"]
```

```
## [1] a b c d e
## Levels: a b c d e
```

```
subset(a_data_frame, y > 0 | z, x)
```

```
##   x
## 1 a
## 2 b
## 3 c
## 4 d
## 5 e
```

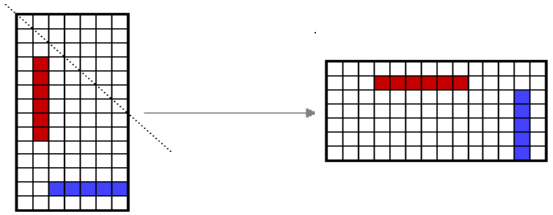
Data frame - combine/transpose/calculation

Combine (for appropriate dimensionality)

- `cbind()`: combine the data frames by column, will not check column names for duplicates
- `rbind()`: combine the data frames by row, it will reorder the columns to match

Transpose

- `t()`: the columns (which become rows) will be converted to the same type



Mean/Sum of a row/column

- `colSums()/colMeans()`: calculate the sums and means of each column
- `rowSums()/rowMeans()`: calculate the sums and means of each row

Image from Internet

Let’s do some practice!

- `git clone https://github.com/wbvguo/qcbio-Intro2R.git`