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

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

Using Base R for your analysis and copy pasting your results into tables in Word

learning how to use dplyr visualize data with ggplot2 and report your analysis in rmarkdown documents

wielding the entire might of the tidyverse

Image from internet

Day 2: R advanced topics

Wenbin Guo

Bioinformatics IDP, UCLA

wbguo@ucla.edu

2024 Spring

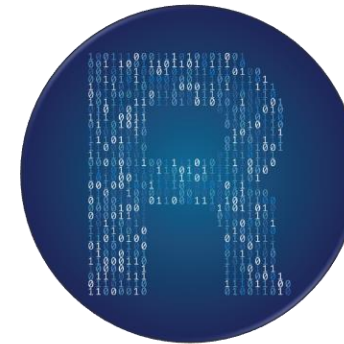# Overview

Time

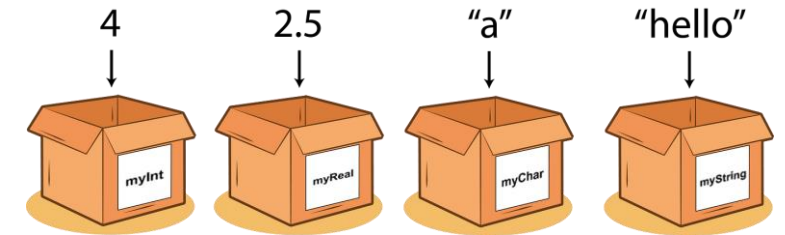- 3-hour workshop (45min + 45min + 30min + practice/Q&A)

Topics

❑Flow control, Loops

❑Function, Packages, File Input/Output

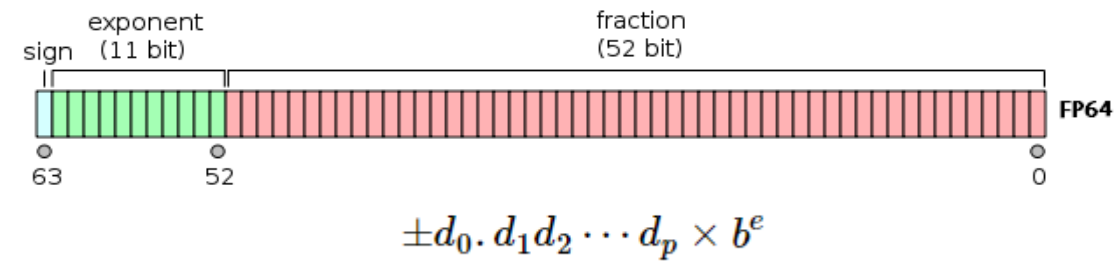❑Data Wrangling with tidyverse toolkit

Summary – Day1

Variables: the container of data

- ❑ Naming rules
- ❑ Value assignment
- ❑ Variable classes
- ❑ Inspecting the variable
- ❑ Inspecting the workspace

Numbers

- ❑ Number representation
- ❑ Special numbers

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

Summary – Day1

Operators: the actions on variables

- ❑ Arithmetic
- ❑ Relational
- ❑ Logical
- ❑ Operator's precedence

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

Operator	Description
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	exactly equal to
!=	not equal to
!x	Not x
x | y	x OR y
x & y	x AND y
isTRUE(x)	test if X is TRUE
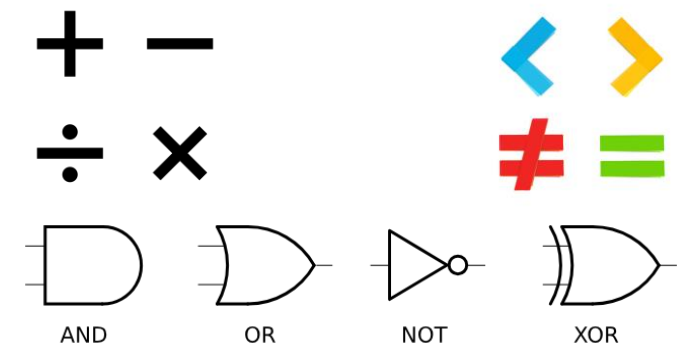
Precedence	Operator	Description
18	:: :::	access variables in a namespace
17	$ @	component / slot extraction
16	[ ] [[ ]]	indexing
15	^	Exponentiation operator (Right to Left)
14	+a -a	Unary plus, Unary minus
13	:	Sequence operator
12	%% %*% %/% %in% %o% %x%	Special operators
11	* /	Multiplication, Division
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)

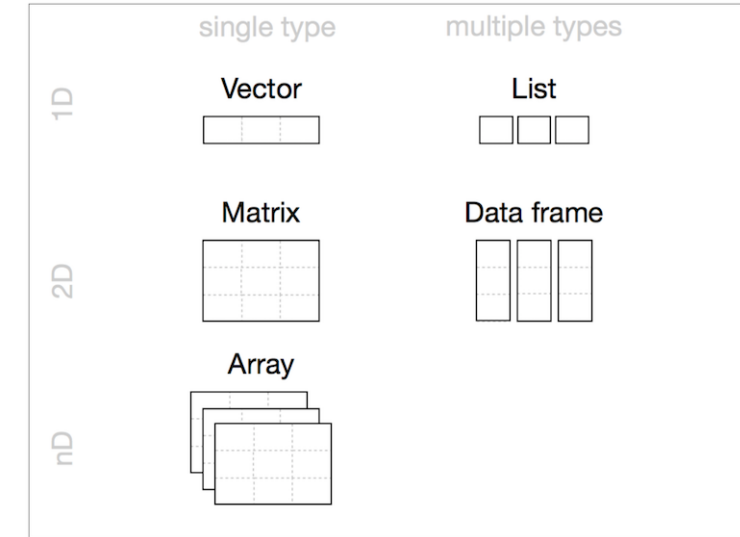Top to bottom in **descending precedence**

Summary – Day1

R objects: data container

- ❑ Vectors
- ❑ String and Factor
- ❑ Matrix
- ❑ List
- ❑ Data frame

Operations on the R objects

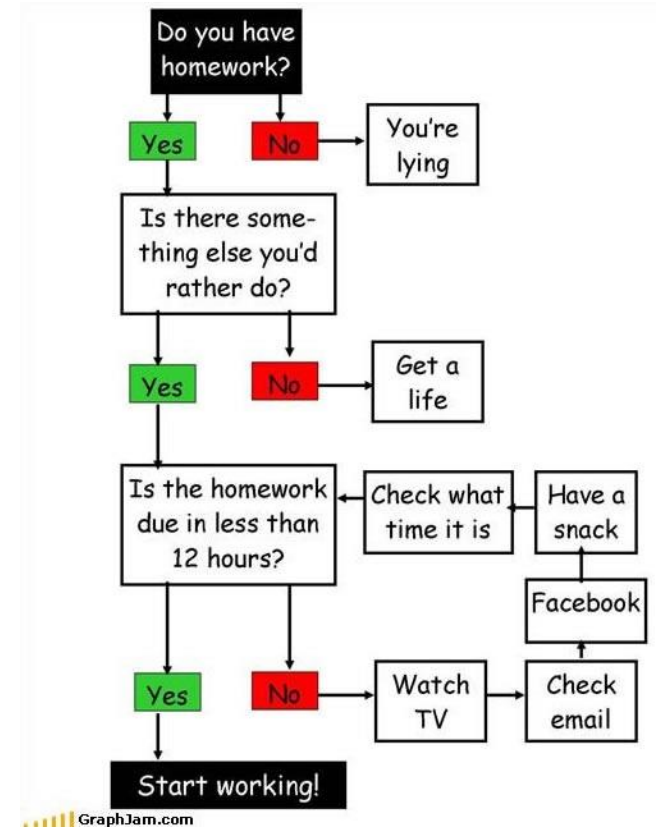- ❑ Create
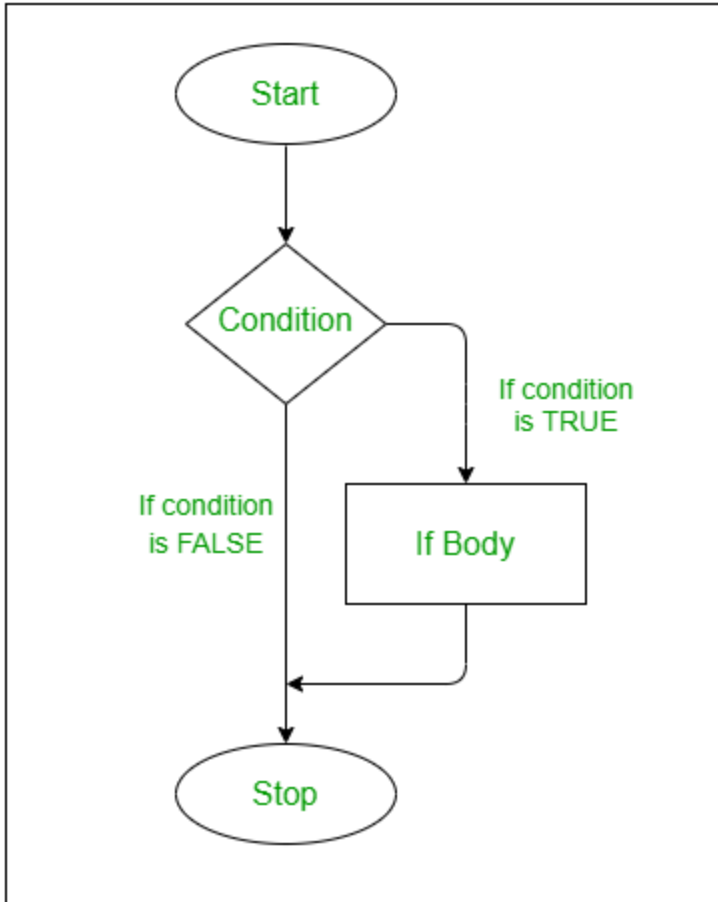- ❑ Indexing
- ❑ Update
- ❑ …

R as a scientific calculator

- ❑ R/RStudio environment setup
- ❑ Get help in R
- ❑ Interesting examples

Flow control

"When in Rome, do as the Romans do."

`if` statement

Condition
Any expression that evaluates to true or false

```
if (condition) {
    statement
    statement
    ...
}
following_statement
```

True branch
This is executed if the condition is true

Syntax:

Example:

```
x <- 3
if(x > 2)
{
    y <- 2 * x
    z <- 3 * y
}
```

Note:
- Condition is a logical value (a logical vector of length one)
- Passing missing value to `if()` is not allowed

if-else statement

Syntax:

```
if (condition) {
    statement          True branch
    statement          This is executed if the
    ...                condition is true
} else {
    statement          False branch
    statement          This is executed if the
    ...                condition is false
}
following_statement
```

Note:
- Condition is a logical value (a logical vector of length one)
- Passing missing value to `if()` is not allowed
- `else` statement must occur on the same line as the closing curly brace from the if clause

if-else if-else statement

Syntax:

```
if (condition) {
    statement                    First condition
    statement            }----→  This is executed if the
                                 first condition is true
    ...
} else if (condition) {
    statement
    statement
New condition
A new condition   ←----
to test if previous
condition isn't true
    ...
} else {
    statement                    False branch
    statement            }----→  This is executed if none
                                 of the conditions are true
    ...
}
following_statement
```


Example:

```
if(is.nan(x))
{
  message("x is missing")
} else if(is.infinite(x))
{
  message("x is infinite")
} else if(x > 0)
{
  message("x is positive")
} else if(x < 0)
{
  message("x is negative")
} else
{
  message("x is zero")
}
```

Stacking **if/else**
statements be like

12

switch statement
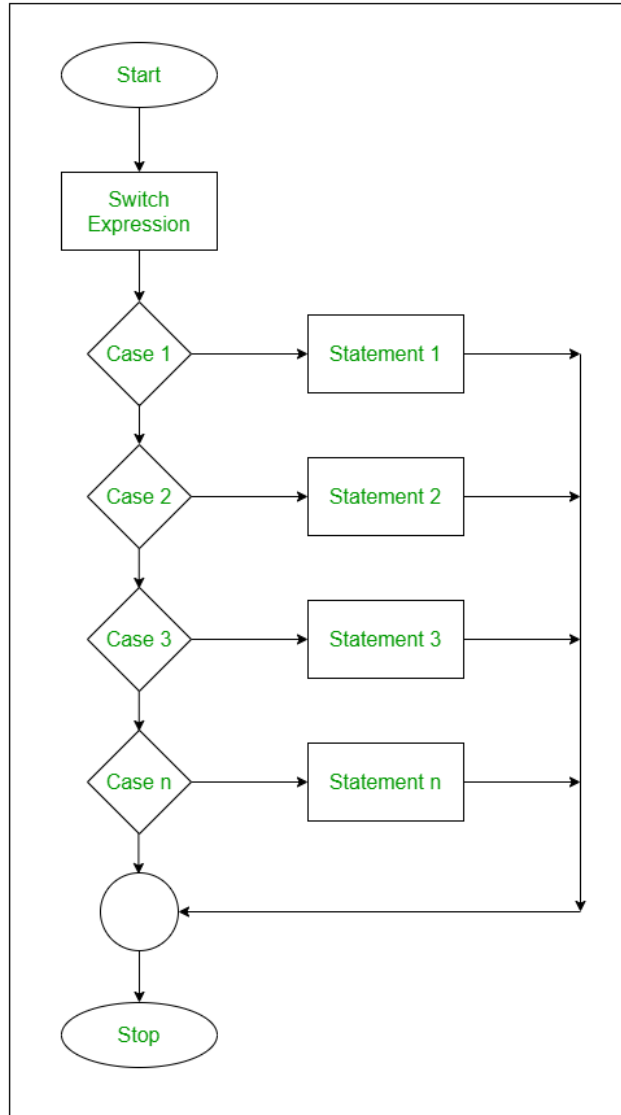
Syntax:

```
variable <- switch(expression, val1, val2, ...)
```

Expression
It can be either a
number or a character

List of values
Value is selected based on
the name or position

Example:

```r
day_of_week_if_else <- function(day) {
  if (day == 1) {
    result <- "Monday"
  } else if (day == 2) {
    result <- "Tuesday"
  } else if (day == 3) {
    result <- "Wednesday"
  } else if (day == 4) {
    result <- "Thursday"
  } else if (day == 5) {
    result <- "Friday"
  } else if (day == 6) {
    result <- "Saturday"
  } else if (day == 7) {
    result <- "Sunday"
  } else {
    result <- "Invalid day"
  }
  return(result)
}
```

```r
day_of_week_switch <- function(day) {
  result <- switch(as.character(day),
                   "1" = "Monday",
                   "2" = "Tuesday",
                   "3" = "Wednesday",
                   "4" = "Thursday",
                   "5" = "Friday",
                   "6" = "Saturday",
                   "7" = "Sunday",
                   "Invalid day")
  return(result)
}
```

13

Other usage

- `if-else` assignment in one line

variable <- if (condition) Statement else Statement;

True branch
Execute this if the condition is true

False branch
Execute this if the condition is false

- Vectorized `if-esle`

ifelse (condition, TrueVector, FalseVector)

Condition
Condition is checked for every element of a vector

True branch
Select element from this if the condition is true

False branch
Select element from this if the condition is false

Let's do some practice!

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

Loops

"History repeats itself."
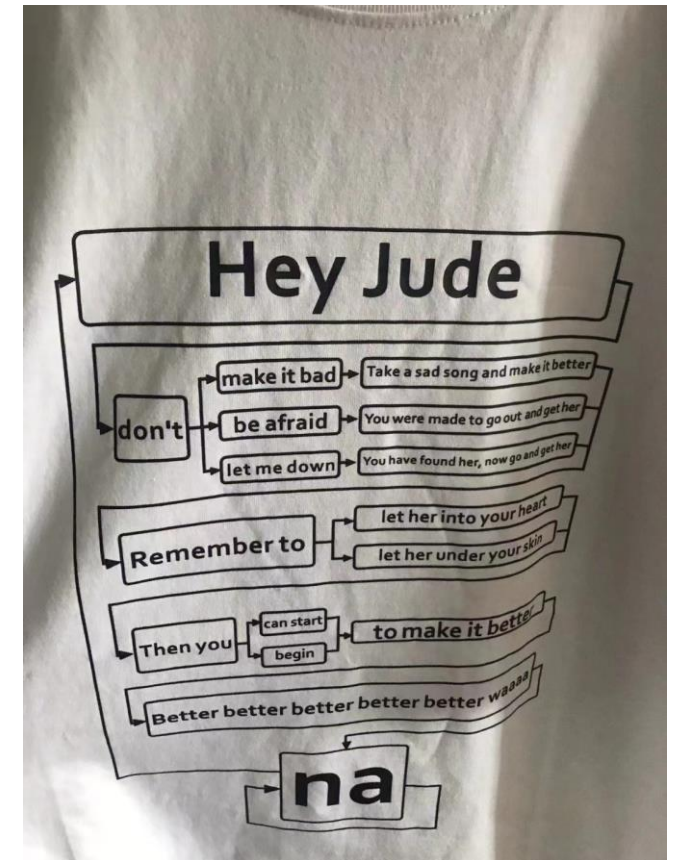
for loop
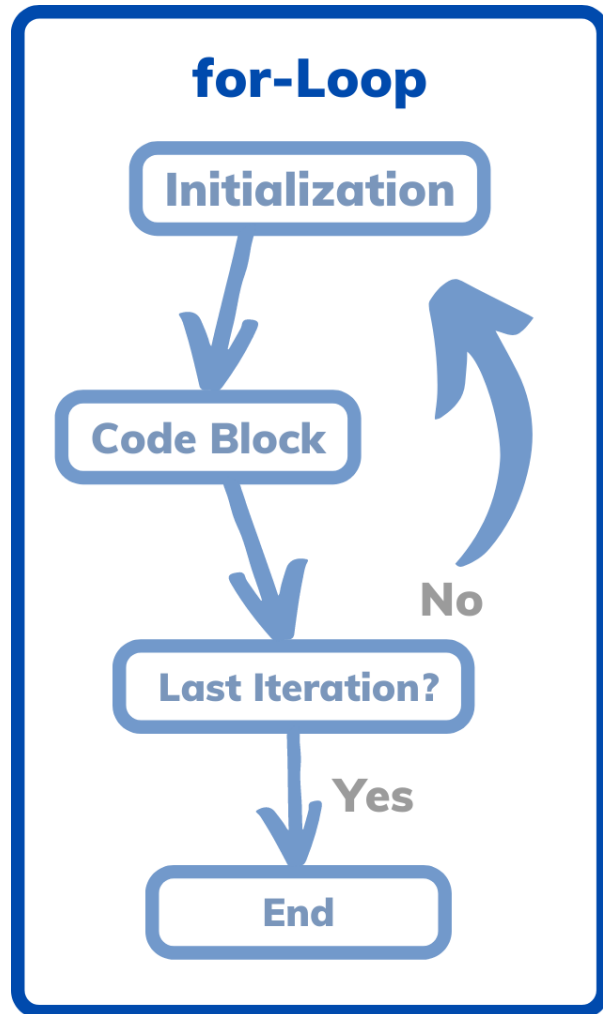
Syntax:

```
        Var                                    Iterable
It takes items from                    It's a collection of objects
iterable one by one                    (like a vector, list etc.)

for (var in iterable) {
    statement
    statement                          Loop body
    ...                                It is executed once for
                                       each item in iterable
}
following_statement
```

Example:

```
for(i in 1:5)
{
    j <- i ^ 2
    message("j = ", j)
}

## j = 1

## j = 4

## j = 9

## j = 16

## j = 25
```

`while` loop
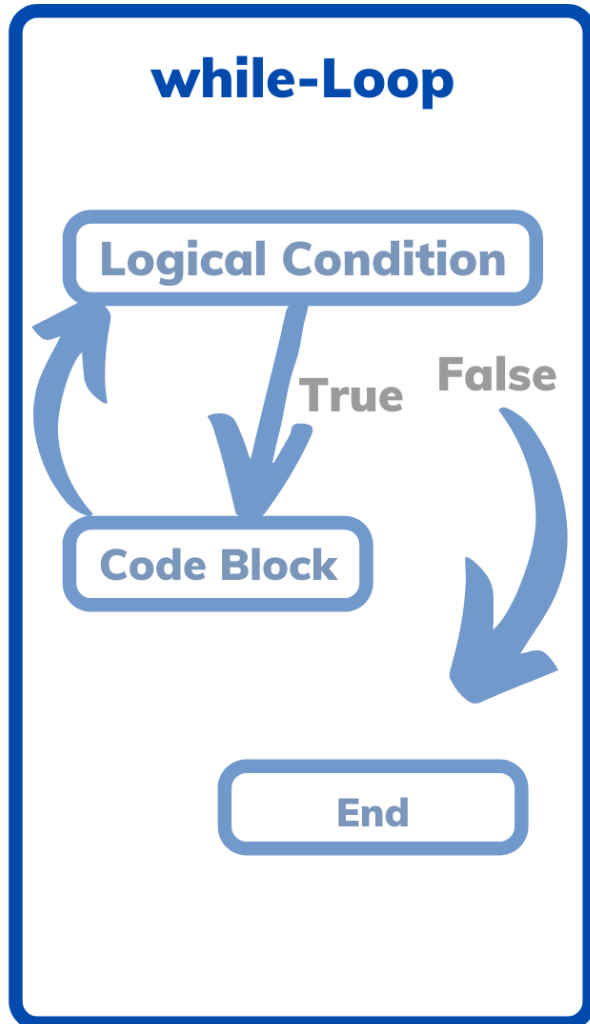
Syntax:

```
                              Condition
                              Any expression that
                              evaluates to true or false

while (condition) {
    statement                 Loop body
    statement                 It is executed as long
                              as the condition is true
    ...
}
following_statement
```

Example:

```
while(alive)
{
    eat();
    sleep();
    code();
}
```

Try it out

- Implement previous example using while loop

Repeat
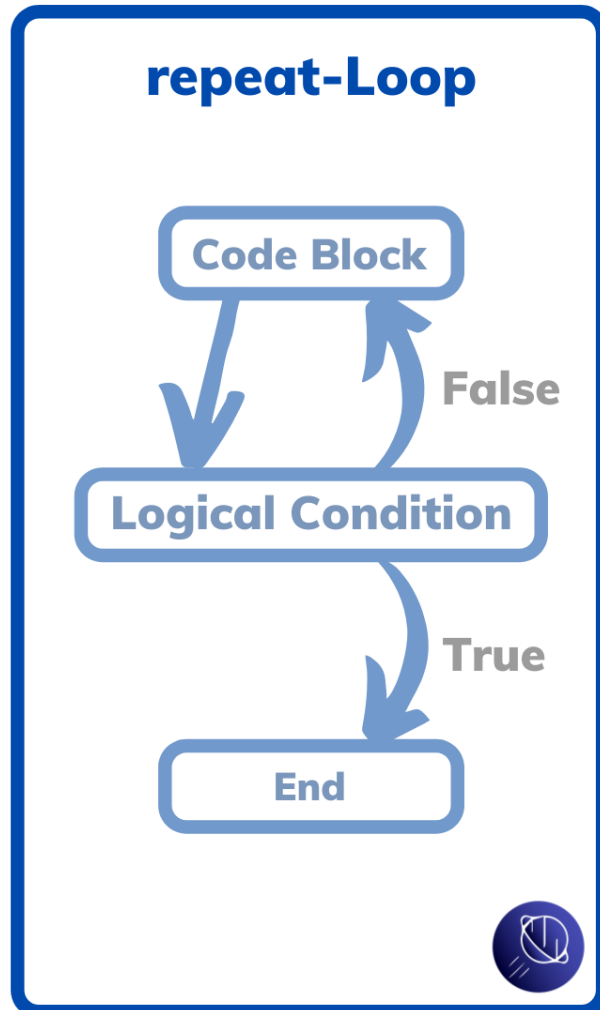
repeat-Loop

Syntax:


```
repeat {
    statement
    statement
    ...
}
following_statement
```

Loop body

Repeat this block of code indefinitely

Try it out

- Implement previous example using repeat loop

The apply family

"There is more than one ways to do it"

- `lapply():` list apply

- `sapply():` simplifying list apply

- `tapply():` table apply

- `apply():` data frame/matrix apply

```
prime_factors <- list(
  two   = 2,
  three = 3,
  four  = c(2, 2),
  five  = 5,
  six   = c(2, 3),
  seven = 7,
  eight = c(2, 2, 2),
  nine  = c(3, 3),
  ten   = c(2, 5)
)
```

for

```
unique_primes <- vector("list", length(prime_factors))
for(i in seq_along(prime_factors))
{
  unique_primes[[i]] <- unique(prime_factors[[i]])
}
names(unique_primes) <- names(prime_factors)
```

apply

```
lapply(prime_factors, unique)
```

There are other functions `mapply()`, `vapply()`…check them out with `?fun_name`

lapply

`lapply()` takes a list and a function as inputs, applies the function to each element of the list in turn, and returns another list of results

$$lapply(x, FUN, …)$$

Parameters

Parameter	Condition	Description
x	Required	A list
FUN	Required	The function to be applied
…	Optional	Any other arguments to be passed to the FUN function

sapply

`sapply()` basically works the same as `lapply()`, but tries to simplify the result as vector or matrix

- if return value is the same length across elements (if length is 1, return vector, else matrix)
- If the return value is not always the same length, return list

`sapply(x,FUN,…)`

Parameters

Parameter	Condition	Description
x	Required	A list
FUN	Required	The function to be applied
…	Optional	Any other arguments to be passed to the FUN function

tapply

`tapply()` breaks the data set up into groups and applies a function to each group

`tapply(x,INDEX,FUN,…,simplify)`

Parameters

Parameter	Condition	Description
x	Required	A vector
INDEX	Required	A grouping factor or a list of factors
FUN	Required	The function to be applied
…	Optional	Any other arguments to be passed to the FUN function
simplify	Optional	Returns simplified result if set to TRUE. Default is TRUE.

apply

The `apply` function provides the row/column-wise equivalent of `lapply()`

`apply(x,MARGIN,FUN,…)`

Parameters

Parameter	Condition	Description
x	Required	A matrix , data frame or array
MARGIN	Required	A vector giving the subscripts which the function will be applied over. 1 indicates rows 2 indicates columns c(1, 2) indicates rows and columns
FUN	Required	The function to be applied
…	Optional	Any other arguments to be passed to the FUN function

Question:

1. how to calculate the row sum of a matrix using apply?

2. How to standardize the rows of a matrix? (mean 0, standard deviation 1)

Let's do some practice!

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

Input

Output

Function

Variables stores data (nouns), functions let us do things with data (verbs)
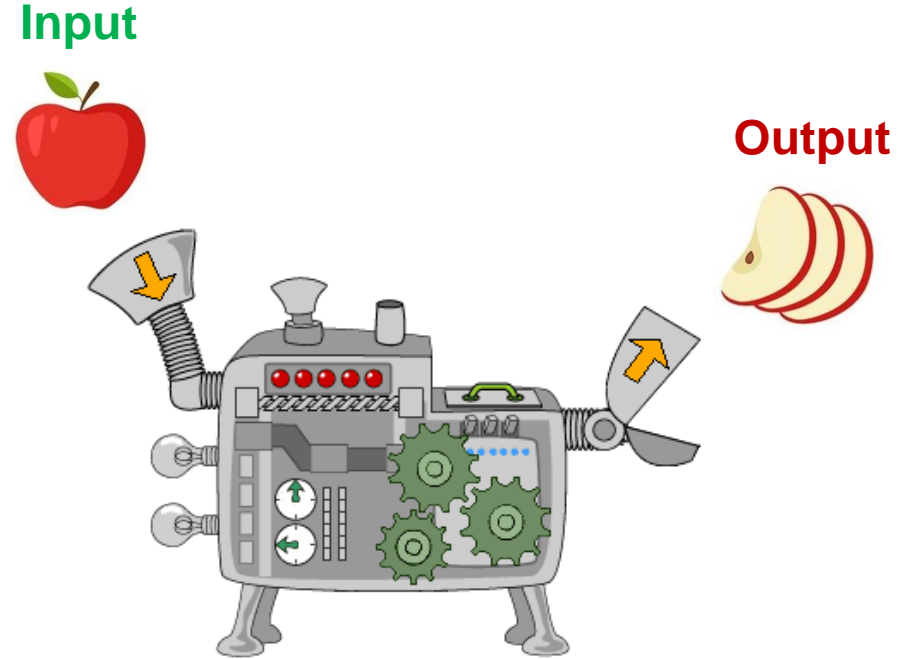
Built-in functions
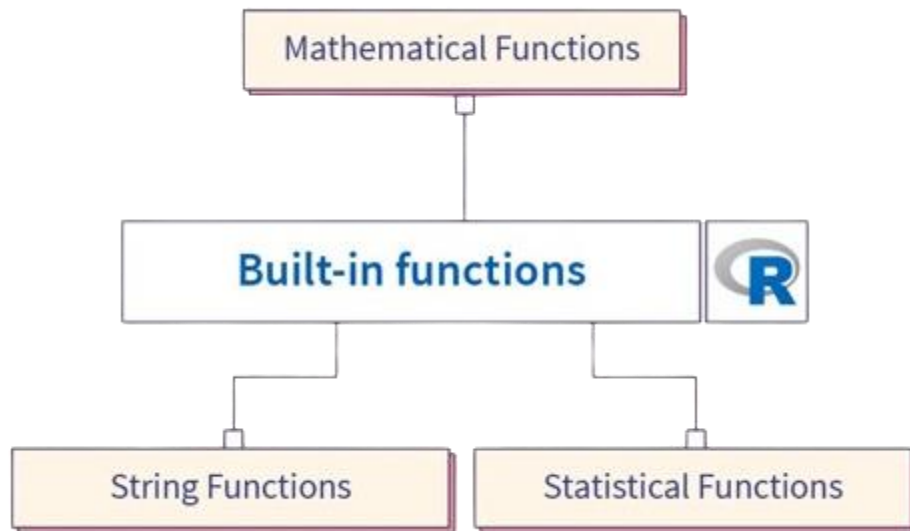
Category	Function Name	Description
Math	abs()	Calculates the absolute value of a numeric vector.
	sqrt()	Computes the square root of each element in a numeric vector.
	sum()	Computes the sum of the vector.
	exp()	Computes the exponential of each element in a numeric vector.
	log()	Computes the natural logarithm of each element in a numeric vector.
Statistical	min()	Calculates the minimum of a numeric vector.
	max()	Calculates the minimum of a numeric vector.
	mean()	Calculates the mean of a numeric vector.
	sd()	Calculates the standard deviation of a numeric vector.
	var()	Computes the sample variance of a given vector.
	cor()	Computes the correlation matrix for numeric variables.
	median()	Computes the median of a numeric vector.
	quantile()	Computes the quantiles of a numeric vector
	rank()	Computes the rank of elements in a numeric vector.
String	paste()	Concatenates strings together.
	toupper()	Converts a character vector to uppercase.
	grep()	Searches for a pattern in a character vector.
	nchar()	Counts the number of characters in each element in a string object.
Other	unique()	Extracts unique elements from a vector.
	sort()	Sorts the elements of a vector in ascending or descending order.
	sample()	Selects random sample elements from a vector.

Customized functions

Function name
An identifier by which the function is called

Arguments
Contains a list of values passed to the function

```
name <- function(args){
    statement
    statement
    ...
    return(value)
}
```

Function body
This is executed each time the function is called

Return value
Ends function call & sends data back to the program
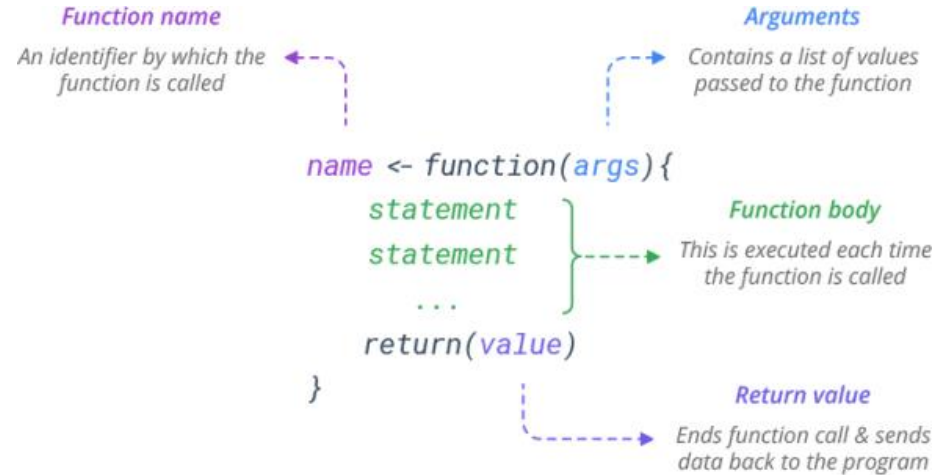
Example:

```
hypotenuse <- function(x, y)
{
  sqrt(x ^ 2 + y ^ 2)
}
```

Function arguments

- Default values can be supplied in function definition: `function(var_name = value){…}`
- Input values are matched to arguments by positions when calling a function without argument name
- R has a special argument ..., that contains all the arguments that aren't matched by position or name

Return

- Without return statement, the last value calculated in the function is automatically returned

Variable scope (local/global)

A variable's scope is the set of places from which you can see the variable

- when you define a variable inside a function (local variable), the rest of the statements in that function will have access to that variable

- Variables defined in the global environment (global variable) can be seen from anywhere

```
h <- function(x)
{
  x * y
}
```

When R tries to find variables

- find variables in the current environment

- if it doesn't find them, it will look in the parent environment, then that environment's parent, and so on recursively until it reaches the global environment

Question: can the global environment access the local variable defined in a function?

Packages

"Code reuse is the Holy Grail of Software Engineering."

R packages: the shoulder of giants

Number of R packages available on CRAN by year
(Excluding archived packages)

Source: CRAN Metadata from {pkgsearch}
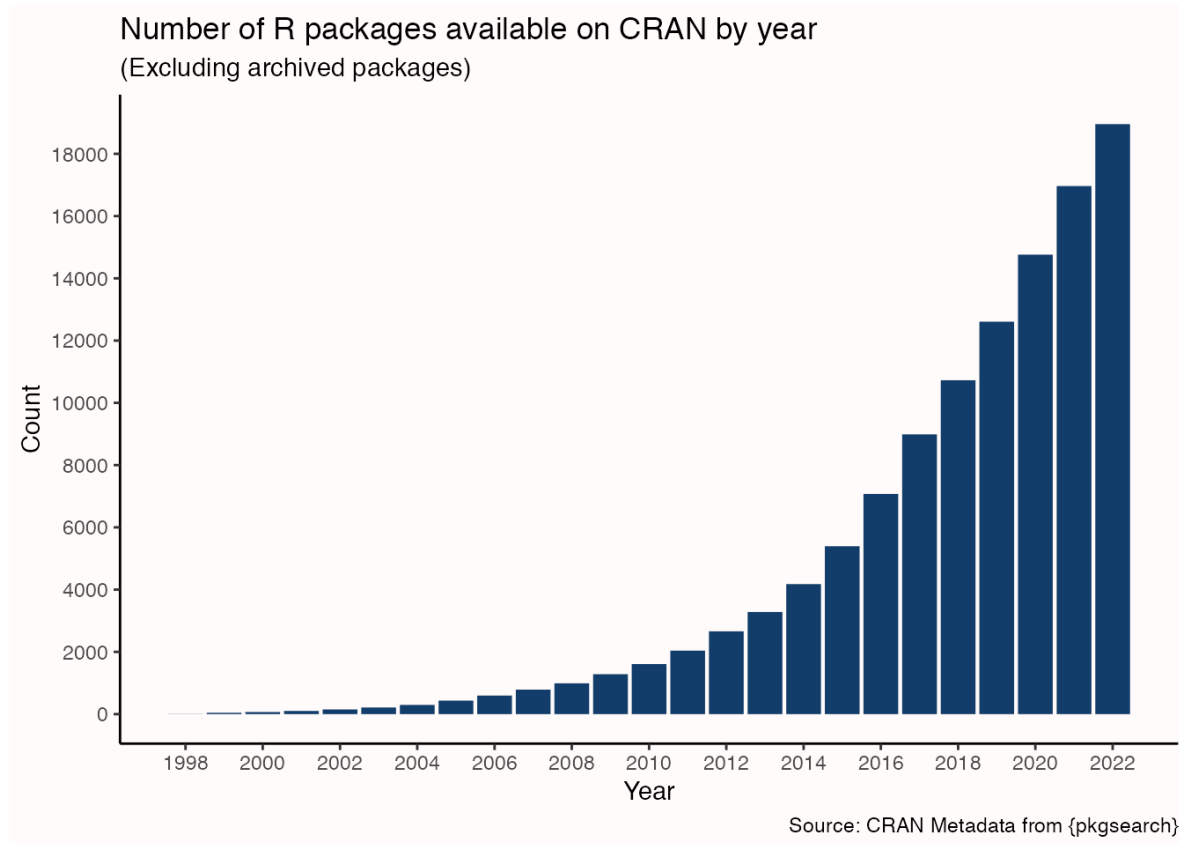
Typically, there is no need to reinvent the wheels…Just realign or improve it if needed

Install / load packages

- Install, update, remove packages
  - ➢ `install.packages("tidyverse")`
  - ➢ `update.packages("tidyverse")`
  - ➢ `remove.packages("tidyverse")`

- Load, unload package
  - ➢ `library(tidyverse)`
  - ➢ `unloadNamespace("tidyverse")`

- Check package's version
  - ➢ `packageVersion("tidyverse")`
  - ➢ `sessionInfo() # print session information and package versions`

Images sourced from https://www.wikihow.com/Change-a-Light-Bulb

Write your own package

- Bundles together code, data, documentation, and tests, in a way that is easy to share.
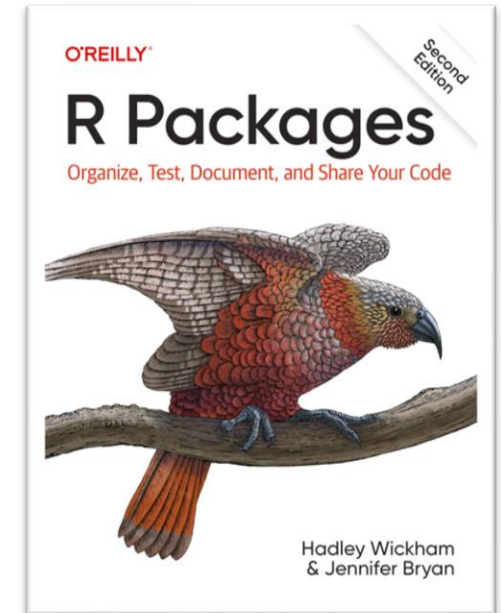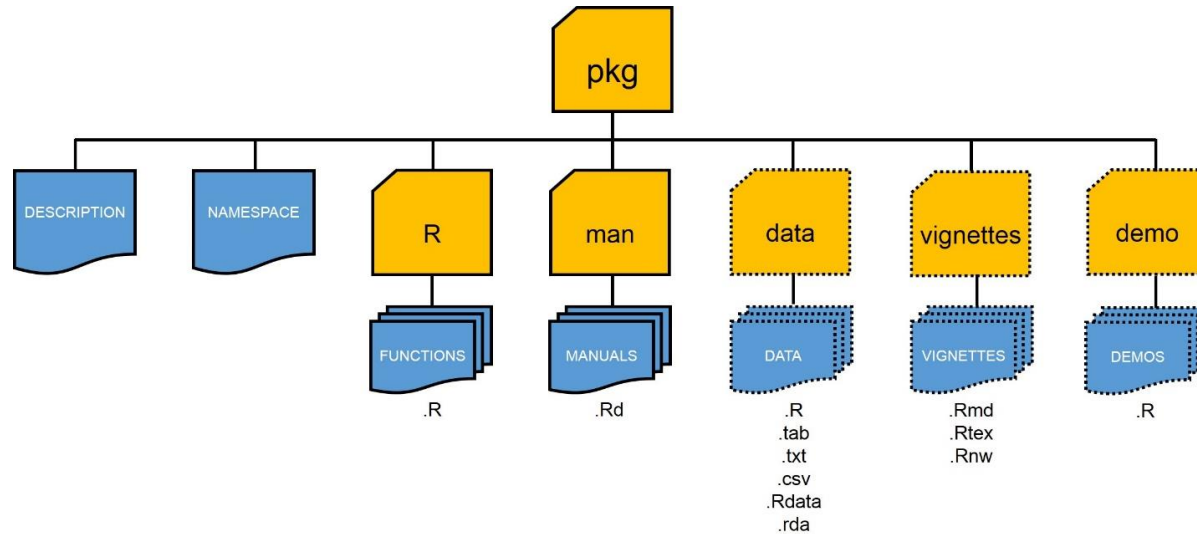
link

Data wrangling
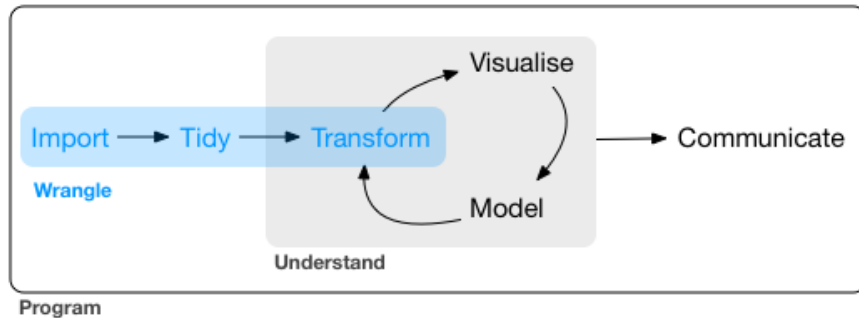
The process of converting raw data into a usable form for modeling/visualization

tidyverse for data wrangling
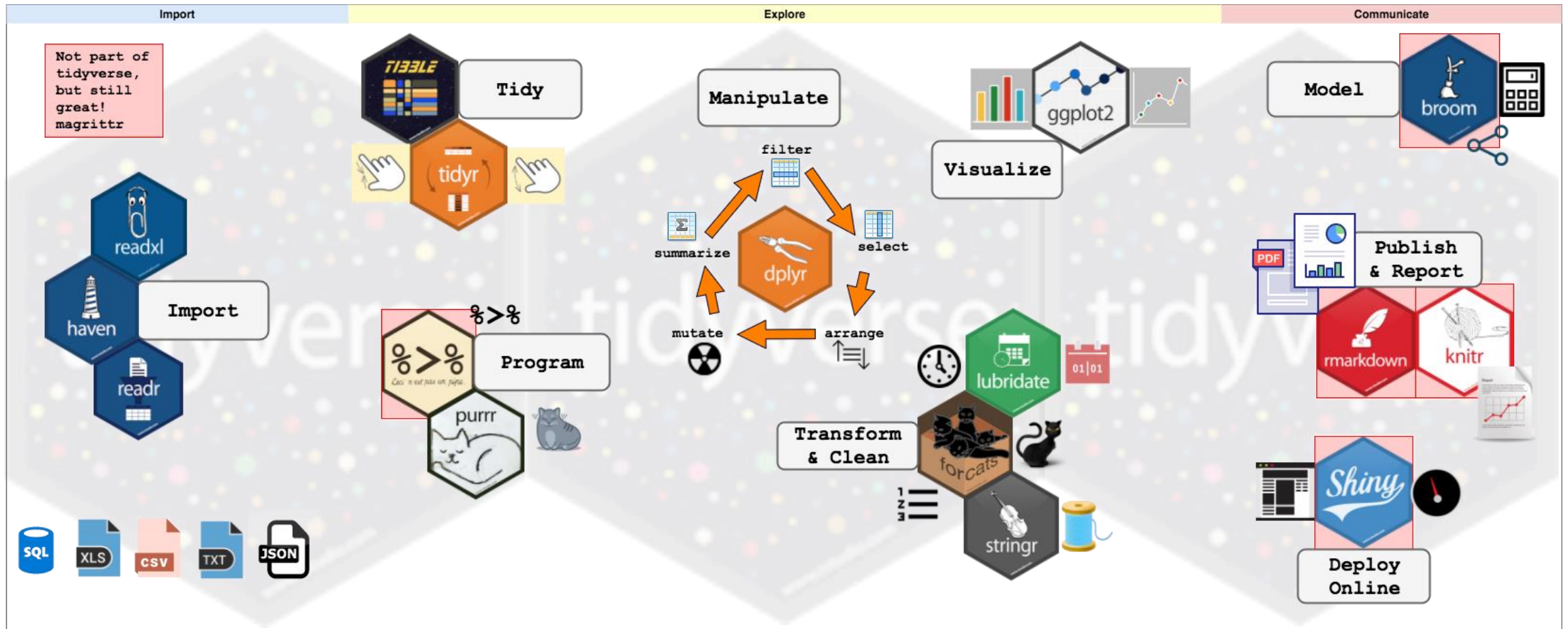
tidyverse for data wrangling

Stackoverflow topics for R

[R] StackOverflow Tags
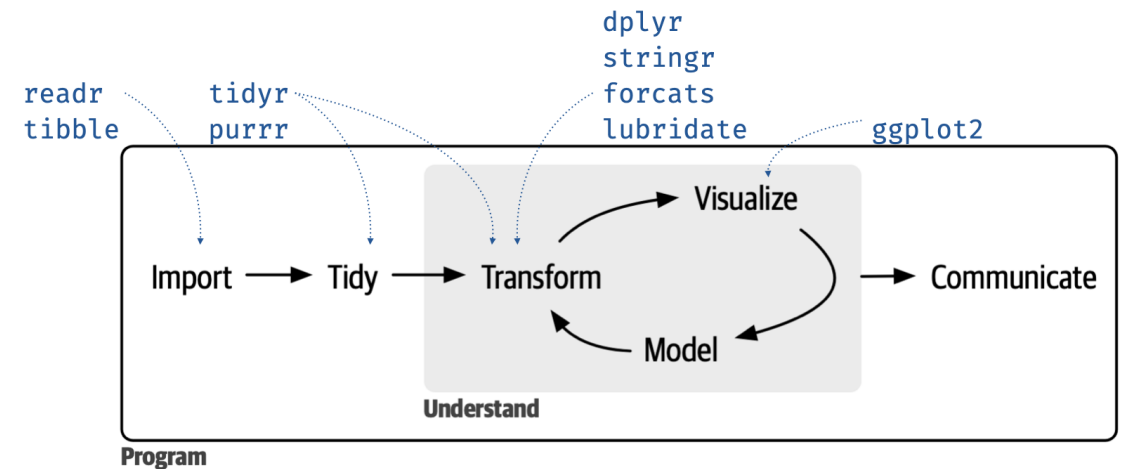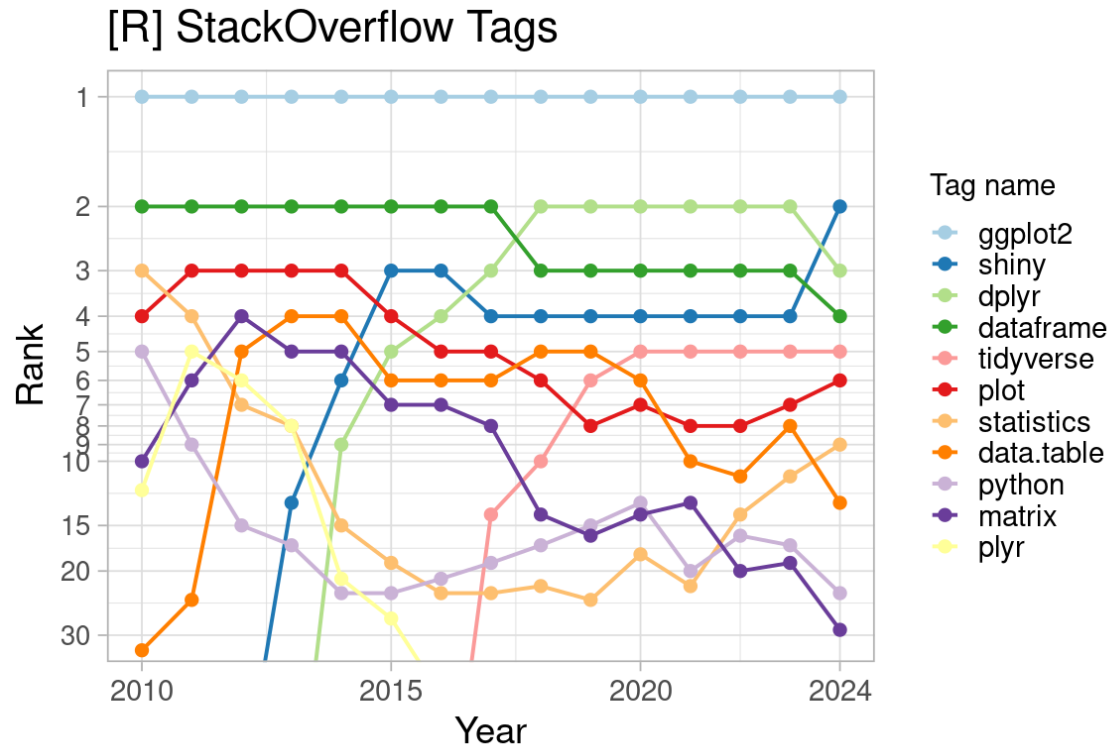
File I/O (.txt/.tsv/.csv)

Read

- `utils::read.table()/read.csv()` (base R)
- `readr::read_table()/read_csv()` (tidyverse)

```
read.table(file, header = FALSE, sep = "", quote = "\"'",
           dec = ".", numerals = c("allow.loss", "warn.loss", "no.loss"),
           row.names, col.names, as.is = !stringsAsFactors,
           na.strings = "NA", colClasses = NA, nrows = -1,
           skip = 0, check.names = TRUE, fill = !blank.lines.skip,
           strip.white = FALSE, blank.lines.skip = TRUE,
           comment.char = "#",
           allowEscapes = FALSE, flush = FALSE,
           stringsAsFactors = default.stringsAsFactors(),
           fileEncoding = "", encoding = "unknown", text, skipNul = FALSE)
```

Some important arguments

- `nrows` : Read only `N` lines (default `-1` ; all).
- `skip` : Skip the first `N` lines (default `0`).
- `strip.white` : Remove leading/trailing white spaces from characters.
- `blank.lines.skip` : Ignore blank lines.
- `fileEncoding` : Character set used for encoding (e.g., `"UTF-8"` , `"latin1"` , ...).
- `text` : Read from a character string rather than a file.
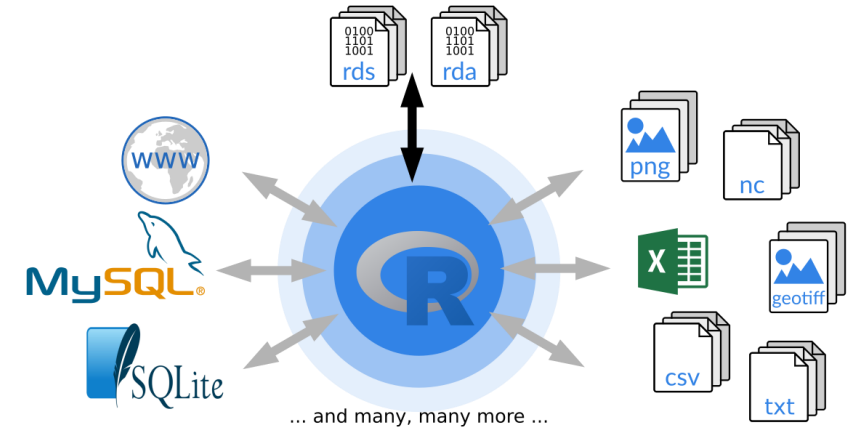
Check out how to write file using ? mark
- `write.table()/write.csv()`
- `write_table()/write_csv()`

37

File I/O (.rds)

Load data

➢ `obj = readRDS("/path/to/file.rds")`

Output data

➢ `saveRDS(obj, "/path/to/file.rds")`

Similar to the `pickle` module in Python

Python Objects   **Binary File**

pickle.dump()

key: value
key: value
key: value

0 1 0 1 1
0 0 1 1 0 0 1
0 1 0
1 1 1
1 1 0 1 0 1

1 | 3 | 5 | 2 | 4

pickle.load()

Objects in Bytes
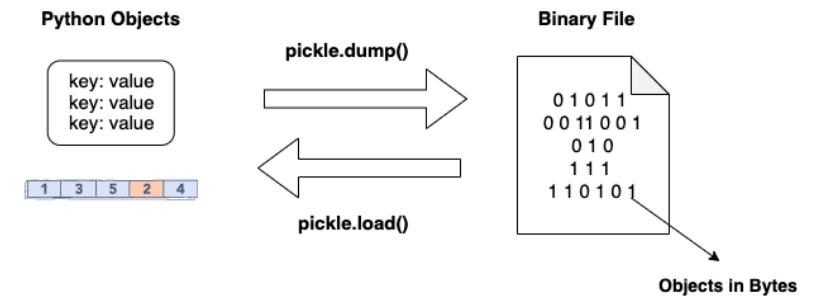
Manipulate Data

"Once you have

❑ the right data,

❑ in the right format,

❑ aggregated in the right way,

the right visualization is often obvious"

Recap: Variables

The container for storing values

- **Categorical variables**: take discrete values
  - ➢ `x = c("apple", "banana")`   # nominal variables: without an order
  - ➢ `y = c("low", "medium", "high")` # ordinal variables: with an order
- **Continuous variables**: take any values within a range
  - ➢ `z = c(0.05, 1, -2)`

Recap: Data Frame

A generic data object that are used to store tabular data
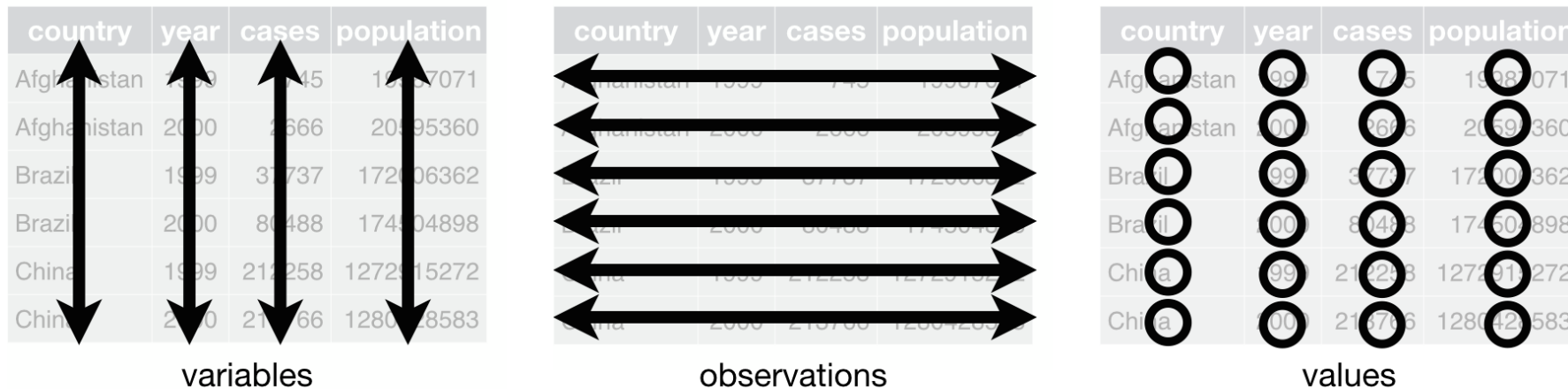
Data wrangling functions

- Manipulate observations (rows)
  - `filter()`
  - `arrange()`
  - `bind_rows()`

- Manipulate variables (columns)
  - `select()`
  - `mutate()`
  - `left_join(), right_join()` …

- Reshape the data
  - `pivot_longer(), pivot_wider()`

- Group and summarize
  - `group_by()`
  - `summarize()`

For more information, check the cheatsheet

Rows

Observations
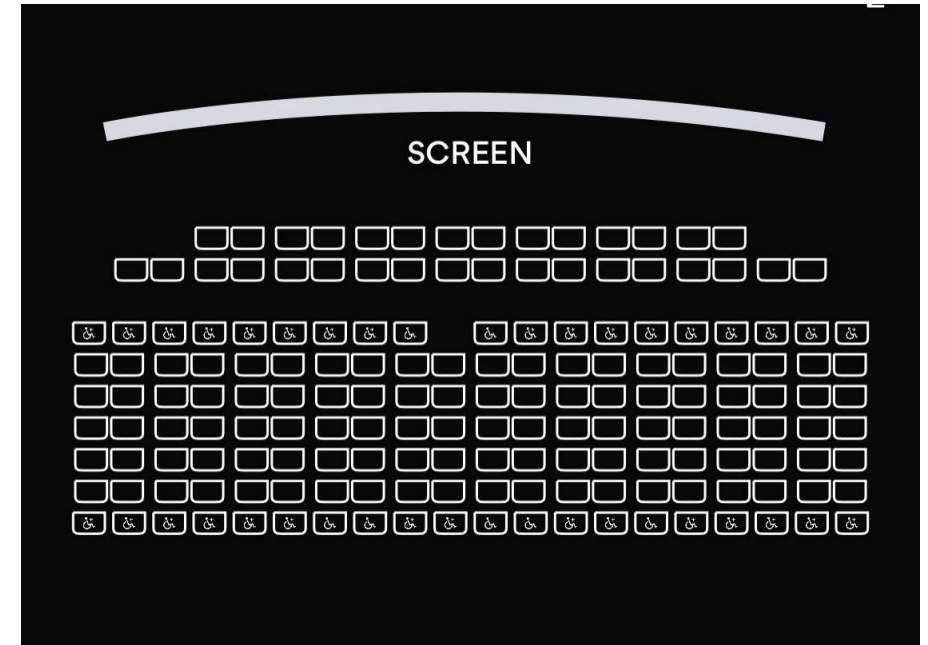
AMC seats

Manipulate observations

`filter()`: keep rows that satisfy certain conditions
- The first argument is a data frame
- The second and subsequent arguments must be logical vectors

Create logical vectors:
- ❏ Comparison operators

  - x == y: x and y are equal.
  - x != y: x and y are not equal.
  - x %in% c("a", "b", "c"): x is one of the values in the right hand side.
  - x > y, x >= y, x < y, x <= y: greater than, greater than or equal to, less than, less than or equal to.

- ❏ Logical operators

  - !x (pronounced "not x"), flips TRUE and FALSE so it keeps all the values where x is FALSE.
  - x & y: TRUE if both x and y are TRUE.
  - x | y: TRUE if either x or y (or both) are TRUE.
  - xor(x, y): TRUE if either x or y are TRUE, but not both (exclusive or).

filter
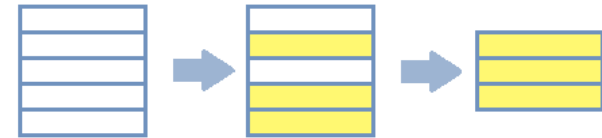
Manipulate observations
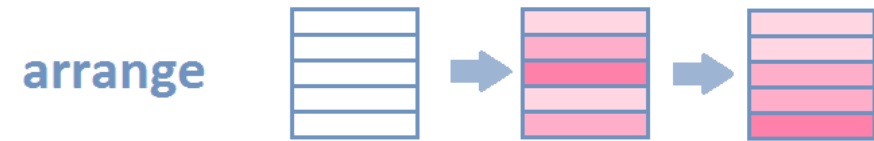
`arrange()`: orders observations according to variables

- The first argument is a data frame
- The second and subsequent arguments are variables or function of variables
- `.by_group`: If `TRUE`, will sort first by grouping variable. Applies to grouped data frames only

arrange

Note:

- the default sorting order is ascending
- use `desc()` to sort a variable in descending order

Manipulate observations

`bind_rows()`: Bind any number of data frames by row

- The first and subsequent arguments are data frames to combine
- Columns are matched by name, and missing columns will be filled with `NA`

`bind_rows()`

Exercise: let's do some practice

Columns

Variables

LACMA lights

Manipulate variables

`select()`: keep or drop variables using their names and types

- The first argument is a data frame
- The second and subsequent arguments are unquoted expressions separated by comma

select ▸▸▸

Useful functions

- `all_of()`: Matches variable names in a character vector
- `starts_with()/ends_with()`: Starts/ends with a substring
- `where()`: Applies a function to all variables and selects those for which the function returns TRUE
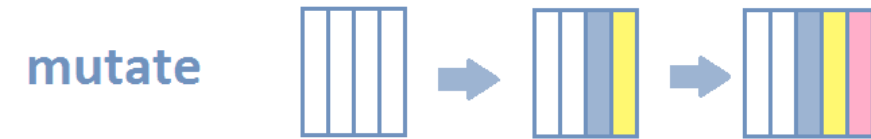
Useful Operators

- `!`: take the complement of a set of variables
- `&` or `|`: select the intersection or union of two sets of variables
- `c()` : combine selections
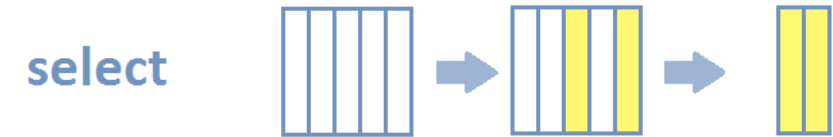
Manipulate variables

`mutate()`: create new variables

- The first argument is a data frame.
- The second and subsequent arguments are name-value pairs (named expression that generate the new variables)
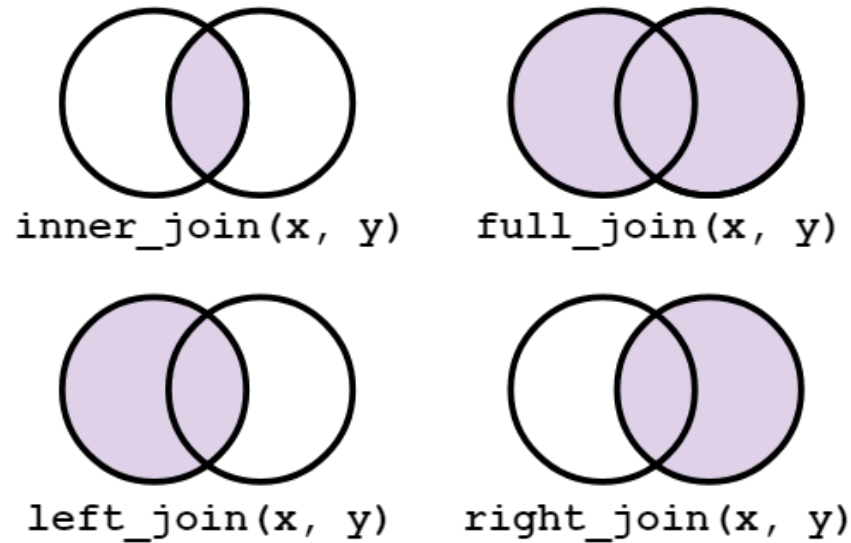

The values can be

- Vector of length 1
- Vector of the same length as whole data frame or current group (for grouped data frame)
- `NULL` to remove the column

Combine datasets

*_join():

inner_join(x, y)

full_join(x, y)

left_join(x, y)

right_join(x, y)

Combine datasets

*_join():

inner_join(x, y)   full_join(x, y)
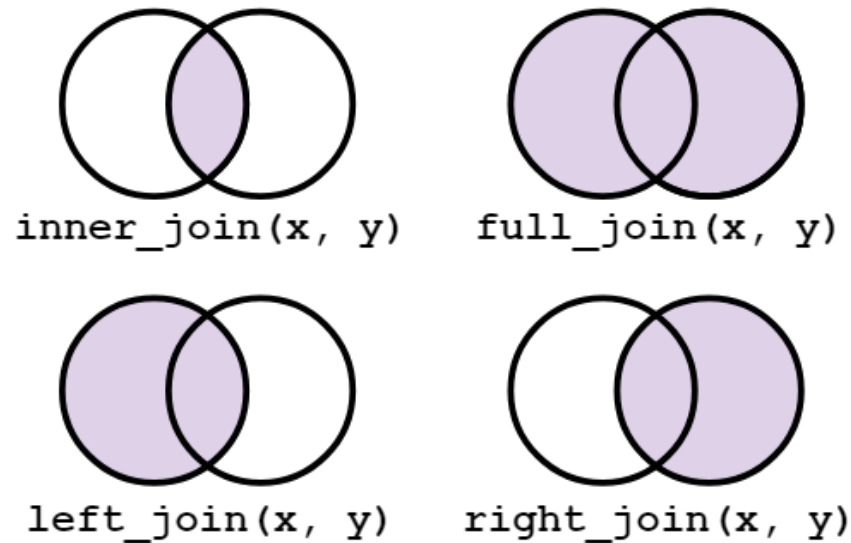
left_join(x, y)   right_join(x, y)

Exercise: let's do some practice

Mutating Joins

dplyr::**left_join(a, b, by = "x1")**
Join matching rows from b to a.

dplyr::**right_join(a, b, by = "x1")**
Join matching rows from a to b.

dplyr::**inner_join(a, b, by = "x1")**
Join data. Retain only rows in both sets.

dplyr::**full_join(a, b, by = "x1")**
Join data. Retain all values, all rows.

Reshape

Landscape ↔ Portrait

Reshape data frame

pivot_longer(): pivot into long format

- The first argument is a data frame
- The second argument is columns to pivot into longer format
- names_to: new column name for column names
- values_to: new column name for cell values

pivot_wider(): pivot into wide format

- The first argument is a data frame
- names_from: column to get the names of output column
- values_from: column to get the cell values

Cartesian data

Indexed data

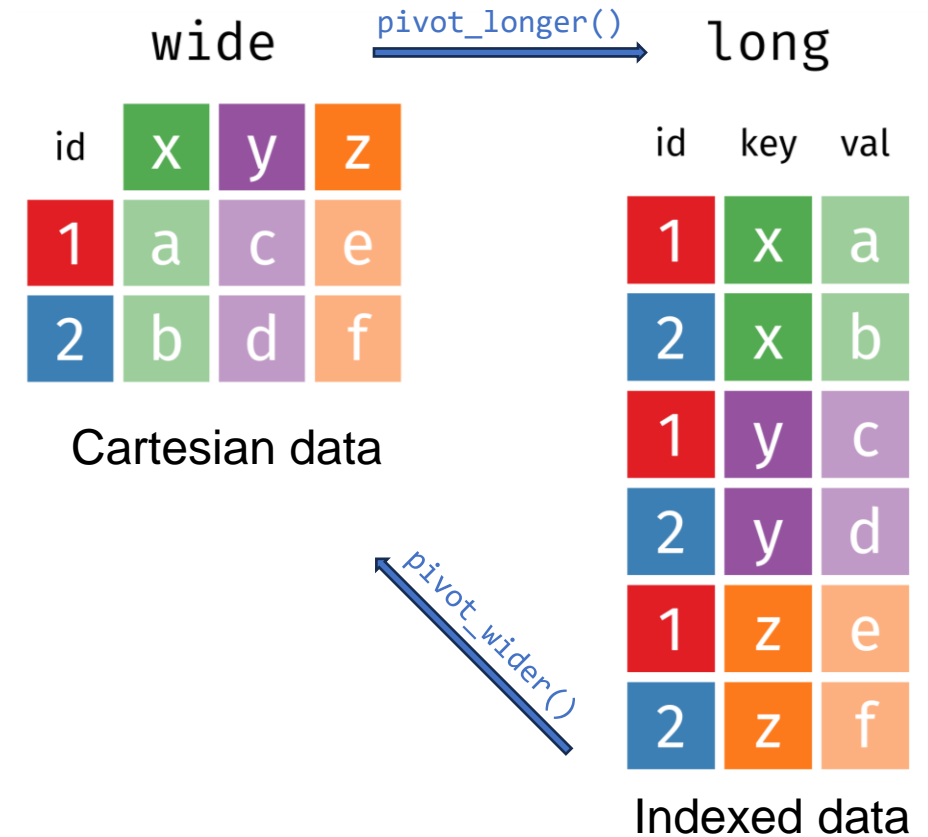https://bookdown.org/Maxine/r4ds/pivoting.html

Reshape data frame

pivot_longer(): pivot into long format

- The first argument is a data frame
- The second argument is columns to pivot into longer format
- names_to: new column name for column names
- values_to: new column name for cell values

pivot_wider(): pivot into wide format

- The first argument is a data frame
- names_from: column to get the names of output column
- values_from: column to get the cell values
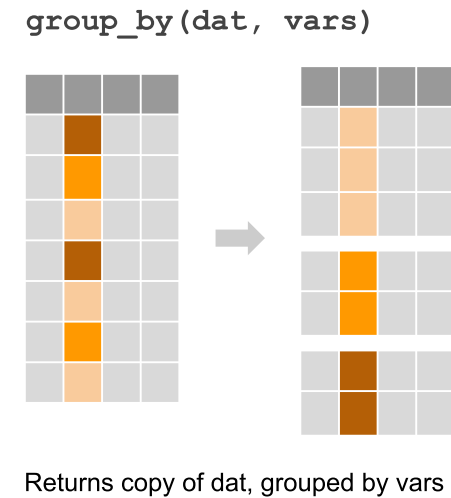
wide

Group

"Birds of a feather flock together."

Group and summarize

group_by(): Define the grouping variables

- The first argument is a data frame
- The second and subsequent arguments are variables used for grouping

Returns copy of dat, grouped by vars

summarise

summarise()/summarize():

- The first argument is a data frame
- The second and subsequent arguments are name-value pairs for summary function

- Counts: n(), n_distinct(x).
- Middle: mean(x), median(x).
- Spread: sd(x), mad(x), IQR(x).
- Extremes: quartile(x), min(x), max(x).
- Positions: first(x), last(x), nth(x, 2).

ungroup(): takes a data frame and removes the grouping

Pipes

"Coming together is a beginning, working together is success." – Henry Ford

Chain the functions together using pipe (%>%)

```
# By using intermediate values
cut_depth <- group_by(diamonds, cut, depth)
cut_depth <- summarise(cut_depth, n = n())
cut_depth <- filter(cut_depth, depth > 55, depth < 70)
cut_depth <- mutate(cut_depth, prop = n / sum(n))
```

```
# By "composing" functions
mutate(
  filter(
    summarise(
      group_by(
        diamonds,
        cut,
        depth
      ),
      n = n()
    ),
    depth > 55,
    depth < 70
  ),
  prop = n / sum(n)
)
```

```
cut_depth <- diamonds %>%
  group_by(cut, depth) %>%
  summarise(n = n()) %>%
  filter(depth > 55, depth < 70) %>%
  mutate(prop = n / sum(n))
```

Question: Which one do you think is the most elegant?

Chain the functions together using pipe (%>%)

```
# By using intermediate values
cut_depth <- group_by(diamonds, cut, depth)
cut_depth <- summarise(cut_depth, n = n())
cut_depth <- filter(cut_depth, depth > 55, depth < 70)
cut_depth <- mutate(cut_depth, prop = n / sum(n))
```

```
# By "composing" functions
mutate(
  filter(
    summarise(
      group_by(
        diamonds,
        cut,
        depth
      ),
      n = n()
    ),
    depth > 55,
    depth < 70
  ),
  prop = n / sum(n)
)
```

```
cut_depth <- diamonds %>%
  group_by(cut, depth) %>%
  summarise(n = n()) %>%
  filter(depth > 55, depth < 70) %>%
  mutate(prop = n / sum(n))
```

%>% works by taking the object on the left hand side (LHS) and using it as the first argument to the function on the right hand side (RHS)

➢ `f(x,y)  <=>  x %>% f(y)`

Question: how to rewrite `g(f(x, y), z)` using pipe?

59

Let's do some practice!

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