Probability - Tidyverse 2[1]

functions, ~~coding practice~~, ~~data structure~~ and introducton to `purrr`

Introduction to Quantitative Social Science

Xiaolong Yang

University of Tokyo

June 28, 2022

[1] Sources: R for Data Science; Advanced R by Hadley Wickham

Today's Game Plan

1. reducing duplication: **functions**
2. data structure: **vector**
3. `purrr` package

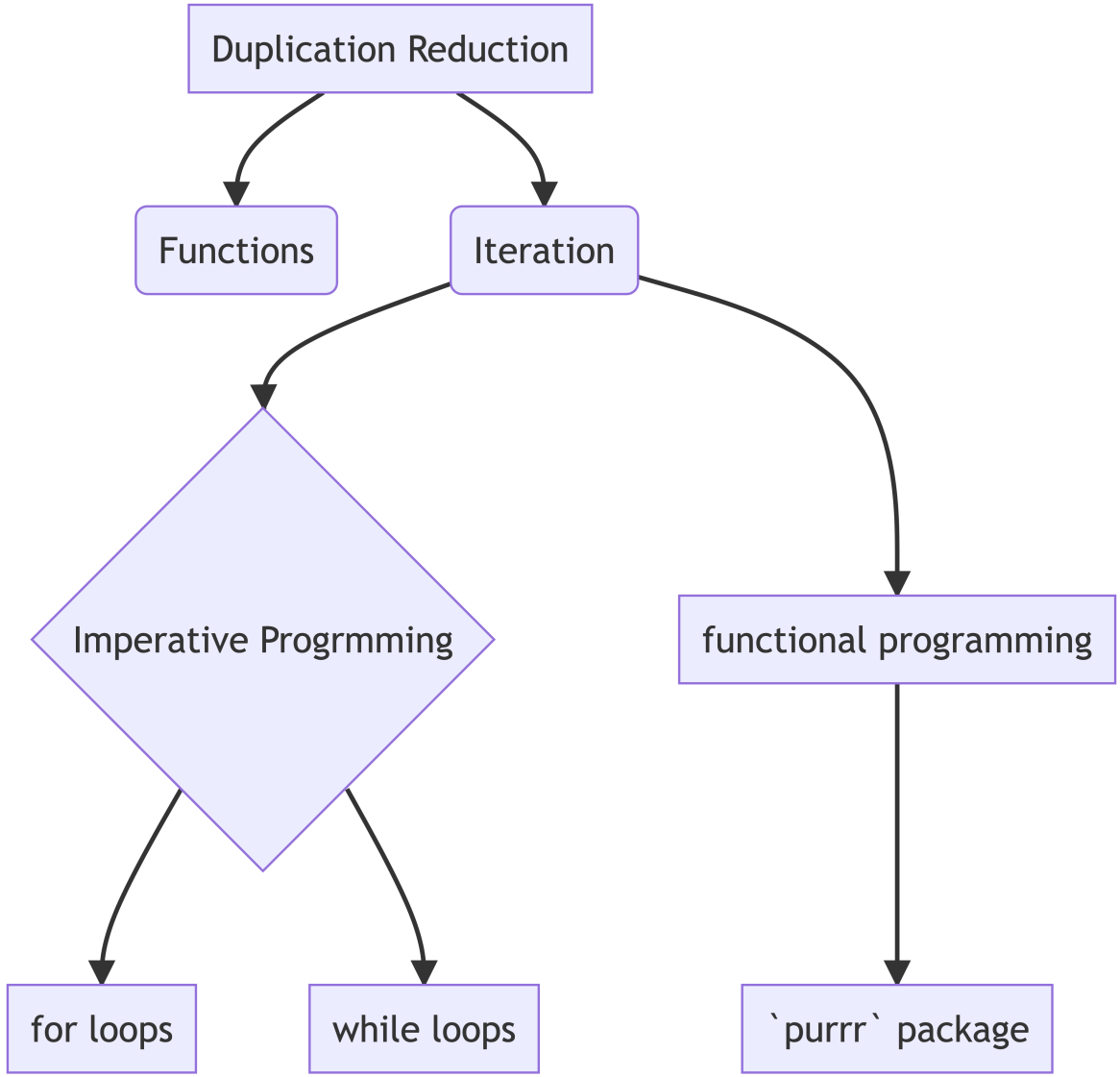
- `purrr`: `map_df()` (introduced in Chapter 6: Probability (sections 6.3-6.4))

> Today's in-class assignment: `intrade-prob`

Section 1

Functions

Landscape of Duplication Reduction in R



Advantages of functions over copy-paste

1. easier to see the intent of your code: eyes on **difference** not **similarity**
2. easier to respond to changes in requirements
3. fewer bugs (i.e. updating a variable name in one place, but not in another).

You should consider writing a function whenever you've copied and pasted a block of code more than twice (i.e. you now have three copies of the same code).

3 key steps to create a function

1. pick a name for the function

```
square
```

3 key steps to create a function

1. pick a name for the function
2. list the inputs, or arguments, to the function inside function

```
square <- function(x) {}
```

3 key steps to create a function

1. pick a name for the function
2. list the inputs, or arguments, to the function inside function
3. place the code you have developed in body of the function

It's easier to start with working code and turn it into a function; it's harder to create a function and then try to make it work

```
square <- function(x) {
  x^2
}

square(13^7)
```

```
[1] 3.937376e+15
```

Function arguments

- data arguments: come first
- details arguments: go on the end

```
# Compute confidence interval around mean using normal approxi
mean_ci <- function(x, conf = 0.95) {
  se <- sd(x) / sqrt(length(x))
  alpha <- 1 - conf
  mean(x) + se * qnorm(c(alpha / 2, 1 - alpha / 2))
}

x <- runif(100)
mean_ci(x)
#> [1] 0.4976111 0.6099594
mean_ci(x, conf = 0.99)
#> [1] 0.4799599 0.6276105
```

Conditional execution

- an `if` statement allows you to conditionally execute code

```
if (condition) {
  # code executed when condition is TRUE
} else {
  # code executed when condition is FALSE
}
```

Multiple conditions

- chain multiple `if` statements together

```
if (this) {
  # do that
} else if (that) {
  # do something else
} else {
  #
}
```

~~Good practices~~

- function names: `verbs`
- argument names: `nouns`
- ~~snake_case~~
- explain the "why", avoid the "what" or "how"

```
# Too short
f()

# Not a verb, or descriptive
my_awesome_function()

# Long, but clear
impute_missing()
collapse_years()
```

```
# Good
input_select()
input_checkbox()
input_text()

# Not so good
select_input()
checkbox_input()
text_input()
```

Section 2

Prerequisite for iteration: Data structure

Visualizing Vectors: *2 types of vector in R*

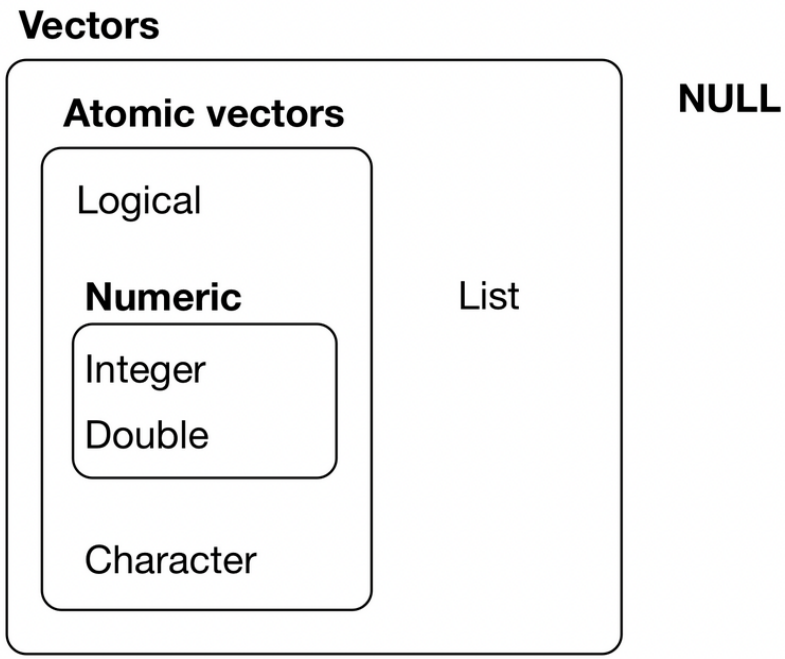


Figure 1: The hierarchy of R's vector types; source: R4DS

Vectors: *2 types of vector in R*

Atomic Vectors

- logical (TRUE/FALSE)
- numeric (integer, double)
- character

check properties: `typeof()` and `length()`

```
x <- c(TRUE, TRUE, FALSE)


typeof(x)
```

```
[1] "logical"
```

```
length(x)
```

```
[1] 3
```

Homogeneous: stores only one type of data

Lists

Heterogeneous: stores different types of data

```
x <- list(1,
          c(2, 3),
          "QSS",
          list(4, 5))


str(x)
```

```
List of 4
 $ : num 1
 $ : num [1:2] 2 3
 $ : chr "QSS"
 $ :List of 2
  ..$ : num 4
  ..$ : num 5
```

Visualizing Vectors: *2 types of vector in R*

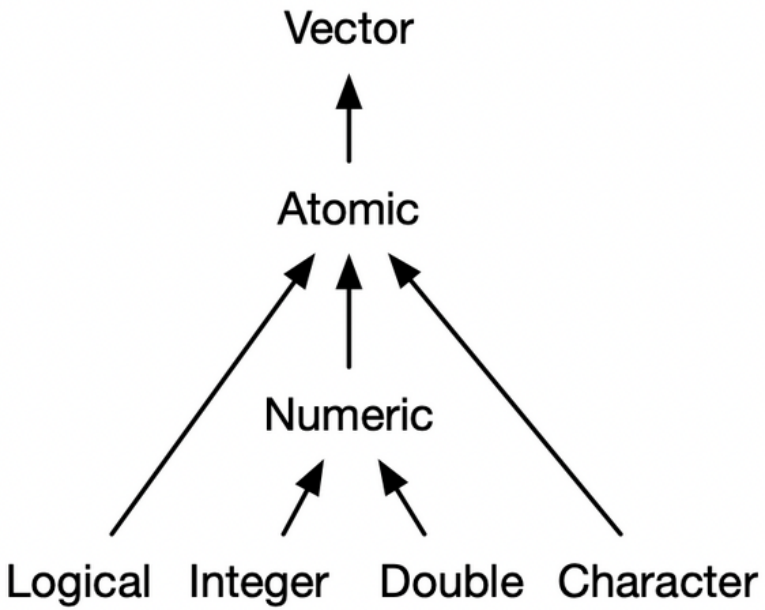


Figure 2: The hierarchy of Atomic vector; source: Advanced R

Visualizing lists

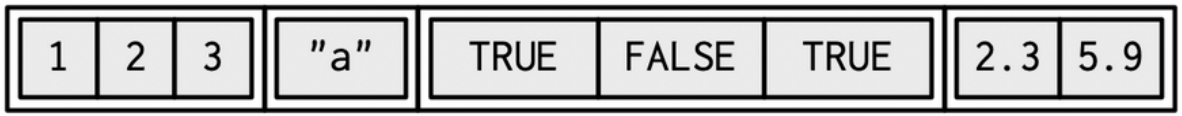


Figure 3: Visualization of a list; source: Advanced R

~~Visualizing lists~~

```
x1 <- list(c(1, 2), c(3, 4))
x2 <- list(list(1, 2), list(3, 4))
x3 <- list(1, list(2, list(3)))
```

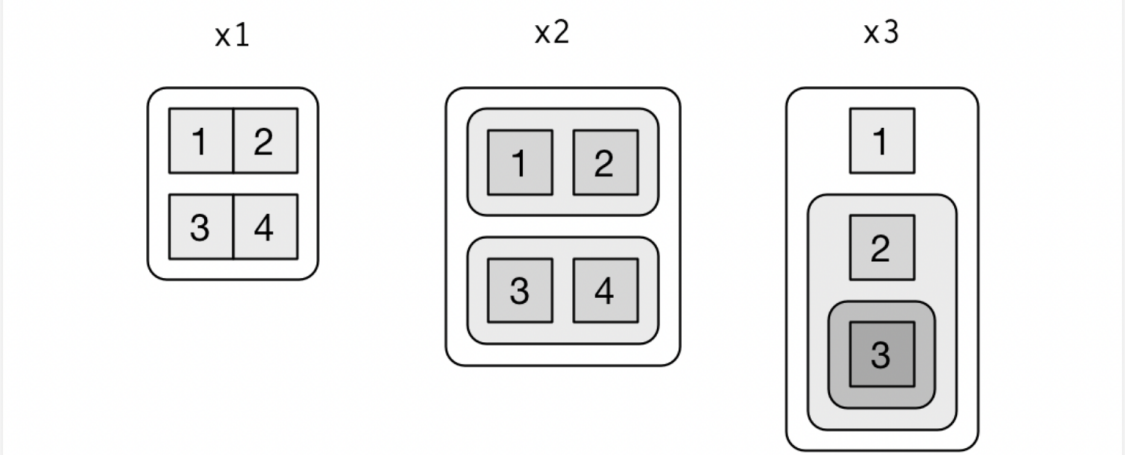


Figure 4: Visualization of lists; source: R4DS

Test functions

- `in_logical()`
- `is_integer()`
- `is_double()`
- `is_numeric()`
- `is_character()`
- `is_atomic()`
- `is_list()`
- `is_list()`

Good additional resources on R data types by Jenny Bryan **Vectors and lists** and **R objects and indexing**

Data frames/tibbles

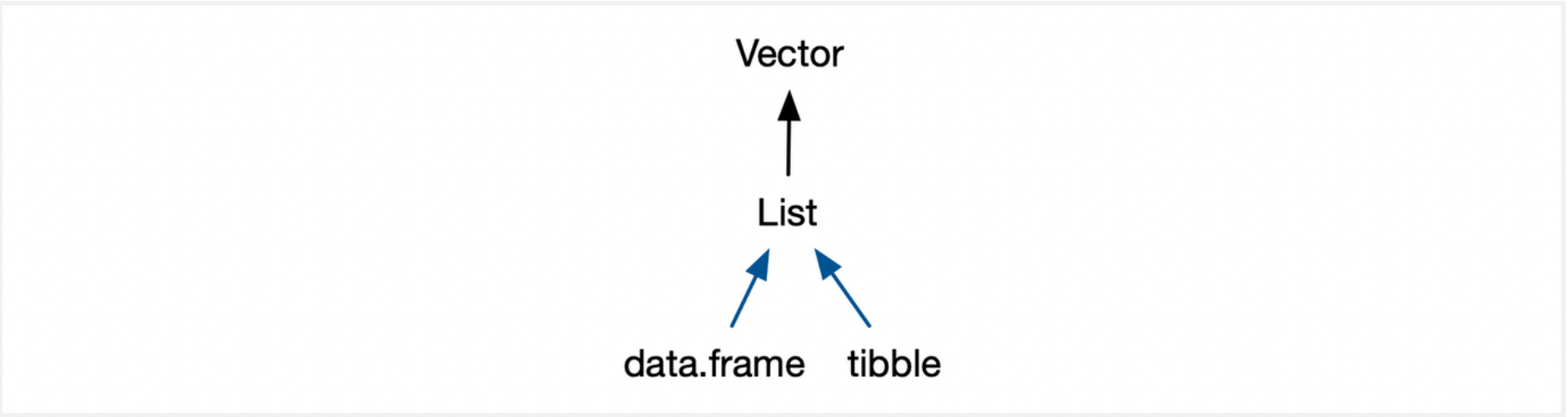


Figure 5: Visualization of data.frame and tibble as lists; source: Advanced R

data frames/tibbles

A data frame is a **named list**, but with specific constraint

- items in the list = columns
- each item (column) = a vector
- every vector (column) has the same length (observations)
- rows = nth items from each vector (nth observations)

```
class(FLVoters)
```

```
[1] "data.frame"
```

```
typeof(FLVoters)
```

```
[1] "list"
```

```
length(FLVoters)
```

```
[1] 6
```

```
str(FLVoters)
```

```
'data.frame':   10000 obs. of  6 variabl
 $ surname: chr  "PIEDRA" "LYNCH" "CHEST
 $ county : int  115 115 115 115 115 115
 $ VTD    : int  66 13 103 80 8 55 84 48
 $ age    : int  58 51 63 54 77 49 77 34
 $ gender : chr  "f" "m" "m" "m" ...
 $ race   : chr  "white" "white" NA "whi
 - attr(*, "spec")=
  .. cols(
```

Short Summary: data structure in R

- vector as the most important data type
  - atomic vector
  - list

Section 3

Brief introduction to purrr

Overview

- R is a functional programming (FP) language
- `purrr` provides complete and consistent tools for working with **functions** and **vectors** $\rightarrow$ enhances R's FP
  - the family of `map()` function $\rightarrow$ replace many for loops with succinct code

purrr package: map_df()

- transform the input by applying a function to **each element of a list** or **atomic vector**
- returns a data frame ~~by row-binding the individual elements~~
- arguments
  - `.x` = a list of atomic vector
  - `.f` = a function, formula, or vector

```
map_df(.x, .f)
```

purrr package: map_df()

```
FLVoters %>%
  map_df(class)
```

```
# A tibble: 1 x 6
  surname   county  VTD     age     gender    race
  <chr>     <chr>   <chr>   <chr>   <chr>     <chr>
1 character integer integer integer character character
```

```
class(FLVoters)
```

```
[1] "data.frame"
```

purrr package: map_df()

```
FLVoters %>%
  map(unique) %>%
  map_df(length)
```

```
# A tibble: 1 x 6
  surname county   VTD   age gender  race
    <int>  <int> <int> <int>  <int> <int>
1    6240     67  1053    82      3     7
```

Summary

What we learnt

- writing functions
- data structure
- `purrr` (`map_df()`)

Future Game Plan

- reducing duplication: **iteration**
- ~~R as **functional programming** language~~
- new functions in **Chapter 7: Uncertainty**