base R, pt.2

Рідлист

- Impatient R (Patrick Burns) — якщо все, що було описано у минулій та цій презентації залишається криптичним та незрозумілим, максимально стисло та просто про самі основи
- The R Inferno (Patrick Burns, 2011) — про common R pitfalls, лінк на .pdf файл

Вектори TL;DR

Вектор є базовим типом структури у R. Скаляри насправді є вектором з довжиною один.

Вектори бувають:

- атомарні, що містять гомогенні **дані одного** типу
  - `character` — текстові значення, мають бути заключені у " " або ' '
  - `logical` — логічні значення
  - `integer` — цілочислові значення
  - `double` — число з плаваючою комою (у панелі середовища відображено як `numeric`)
  - `complex` — комплексне число типу n + i
  - `raw` — для представлення "сирої" послідовності байтів
- листи, що можуть містити гетерогенні **дані різних** типів та інші листи

Створити пустий вектор можливо командою `vector()` або командою, що відповідає одному з типів описаних вище, визначити тип командою `typeof()`, визначити довжину командою `length()`

```
x <- vector(mode = "logical", length = 0L) # теж саме що x <- logical(length = 0L)
typeof(x)
#> [1] "logical"
length(x)
#> [1] 0
```

Чотири основні типи векторів, **`complex`** та **`raw`** зустрічаються відносно рідко і не показані тут

```r
char <- c("This is a String", "This is a second String")
print(char)
#> [1] "This is a String"        "This is a second String"
typeof(char)
#> [1] "character"
length(char) # зверніть увагу на довжину, вектор містить два елементи
#> [1] 2

int <- 1:3
typeof(int)
#> [1] "integer"
also_int <- c(1L, 2L, 3L) # суфікс L експліцитно означає цілочислові значення
typeof(also_int)
#> [1] "integer"

dbl <- c(1.00, 2.5, 0.0002)
typeof(dbl)
#> [1] "double"
also_dbl <- c(1, 2, 3)
typeof(also_dbl)
#> [1] "double"

logic <- c(TRUE, FALSE) # R також допускає скорочення T та F
typeof(logic)
#> [1] "logical"
```

Оскільки атомарні вектори можуть містити лише об'єкти одного типу, при конкатенації векторів різних типів буде виконана спроба конверсії одного типу у інший

Об'єднання будь-якого типу з типом **`character`** конвертує значення у текстові:

```
new_vector <- c(char, logic, dbl)
print(new_vector)
#> [1] "This is a String"        "This is a second String"
#> [3] "TRUE"                    "FALSE"
#> [5] "1"                       "2.5"
#> [7] "2e-04"
typeof(new_vector)
#> [1] "character"
```

Об'єднання типу **`logical`** з чисельними значеннями конвертує логічні значення у чисельні:

```
c(logic, int)
#> [1] 1 0 1 2 3
c(TRUE, TRUE, 5, FALSE, FALSE)
#> [1] 1 1 5 0 0
```

Об'єднання типу **`integer`** з **`double`** дає **`double`**. Як факт, виконання будь-яких математичних операції між **`integer`** та **`double`** поверне значення у форматі **`double`**:

```
typeof(1L + 2)
#> [1] "double"
```

Окрім того, R здатен зберігати у форматі **`integer`** лише значення від -214 748 3647 і до 214 748 3647, **`x <- 2147483648L`** буде збережено як **`double`** з відповідним попередженням

Спробувати “насильно” перевести вектор з одного типу до іншого, або перевірити чи є вектор вектором конкретного типу можливо командою **`as.*(x, ...)`** та **`is.*(x)`** де `*` замінено на відповідний тип

```r
x <- c(TRUE, FALSE)
is.double(x)
#> [1] FALSE
as.double(x)
#> [1] 1 0

as.logical(c("true", "F", "bbbb")) # у випадку неможливості конверсії дає NA
#> [1]  TRUE FALSE    NA
```

Функції **`numeric()`** та **`as.numeric()`** дають такий же результат як **`double()`** та **`as.double()`**, проте **`is.numeric()`** загалом перевіряє чи може об’єкт бути інтерпретований як чисельний

```r
is.numeric(1)
#> [1] TRUE
is.numeric(1L)
#> [1] TRUE
is.double(1L)
#> [1] FALSE
```

Перетворення логічних векторів

Так як логічні значення `TRUE` та `FALSE` мають загальноприйняті відповідні їм чисельні значення `1` та `0`, значна кількість функцій, що виконують дії над чисельними векторами здатні також приймати вектори логічні, автоматично перетворивши їх

Це дозволяє робити так:

```r
set.seed(5839)
logic <- sample(c(TRUE, FALSE), 623, replace = TRUE)

mean(logic)
#> [1] 0.4847512

sd(logic)
#> [1] 0.500169

sum(logic)
#> [1] 302
```

Іменовані вектори

Елементи у векторах можуть мати імена, які можуть бути присвоєні безпосередньо при створенні

```r
named_vec <- c(a = 1, b = 2, c = 3)
named_vec
#> a b c
#> 1 2 3
```

Або для вже інсуючого вектору, командою `names(x) <-` або `setNames()`

```r
names(named_vec) <- c("a", "b", "c")
named_vec
#> a b c
#> 1 2 3
named_vec <- setNames(named_vec, c("x", "y", "z"))
named_vec
#> x y z
#> 1 2 3
```

Переглянути імена елементів об'єкту, його інші атрибути та структуру можливо наступними командами:

```r
str(named_vec) # структура об'єкту
#>  Named num [1:3] 1 2 3
#>  - attr(*, "names")= chr [1:3] "x" "y" "z"
attributes(named_vec) # атрибути об'єкту
#> $names
#> [1] "x" "y" "z"
attr(named_vec, "names") # тільки імена
#> [1] "x" "y" "z"
```

Листи

Пустий лист бажаної довжини також може бути створений командою тією ж командою `vector()`

```r
empty_ls <- vector(mode = "list")
empty_ls # пустий лист довжиною 0
#> list()
```

Сконструювати лист можливо командою `list(...)`

```r
some_ls <- list(c(1, 2, 3), 5.0, "a")
some_ls
#> [[1]]
#> [1] 1 2 3
#>
#> [[2]]
#> [1] 5
#>
#> [[3]]
#> [1] "a"

other_ls <- list(nums = 1:3, lettrs = c("a", "a", "b"), logic = T)
other_ls
#> $nums
#> [1] 1 2 3
#>
#> $lettrs
#> [1] "a" "a" "b"
#>
#> $logic
#> [1] TRUE
```

Лист може мати лист у середні себе

```r
bottomless_pit <- list(
  another_list = list(
   and_other_list = list(
    and_other_other_list = list(
      and_other_other_other_list = list(
       look_inside = "not bottomless",
       pit = "just regular pit now"
      )))))

bottomless_pit
#> $another_list
#> $another_list$and_other_list
#> $another_list$and_other_list$and_other_other_list
#> $another_list$and_other_list$and_other_other_list$and_other_other_other_list
#> $another_list$and_other_list$and_other_other_list$and_other_other_other_list$look_inside
#> [1] "not bottomless"
#>
#> $another_list$and_other_list$and_other_other_list$and_other_other_other_list$pit
#> [1] "just regular pit now"
```

```r
length(bottomless_pit)
#> [1] 1
str(bottomless_pit)
#> List of 1
#>  $ another_list:List of 1
#>   ..$ and_other_list:List of 1
#>   .. ..$ and_other_other_list:List of 1
#>   .. .. ..$ and_other_other_other_list:List of 2
#>   .. .. .. ..$ look_inside: chr "not bottomless"
#>   .. .. .. ..$ pit        : chr "just regular pit now"
```

Сабсетинг векторів

Сабсетинг — "витягування" певного елементу структури, може бути виконане у шість різних способів із застосуванням трьох різних операторів, `[`, `[[` та `$` та може бути комбіноване з `<-`

Оператори `[[` та `$` повертають єдиний елемент. Оператор `$` не допускається до використання з атомарними векторами, але для інших структур може використовуватися для доступу до єлементу по імені

Найпростіший варіант сабсетингу вектору — по індексу (позиції) елементу, позитивним цілим числом:

```r
vec <- c(2.1, 5.2, 2.3, 1.4)
vec[[3]] # повернути елемент у позиції 3
#> [1] 2.3
vec[3]   # для атомарних векторів [i] та [[i]] фактично дають однаковий результат
#> [1] 2.3

vec[1:3]     # елементи з позиції 1 по 3
#> [1] 2.1 5.2 2.3
vec[c(2, 4)] # елемент у позиції 2 та 4
#> [1] 5.2 1.4

# менш інтуїтивні приклади
vec[c(4, 2, 1)]
#> [1] 1.4 5.2 2.1
vec[c(2, 2, 2, 2, 2)]
#> [1] 5.2 5.2 5.2 5.2 5.2
```

Сабсетинг негативним цілим числом повертає усі елементи крім вказаного:

```r
vec[-3]
#> [1] 2.1 5.2 1.4
vec[-c(1, 3)] # теж саме що vec[c(-1, -3)]
#> [1] 5.2 1.4
vec[-c(1:3)]
#> [1] 1.4
```

Сабсетинг текстом дозволяє витягнути елемент по його імені:

```r
names(vec) <- c("one", "two", "three", "four")

vec[["two"]]
#> [1] 5.2
vec[c("one", "one")]
#> one one
#> 2.1 2.1
```

Сабсетинг логічними значеннями повертає елементи позиція яких відповідає значенню TRUE:

```r
vec[c(TRUE, FALSE, TRUE, FALSE)]
#> [1] 2.1 2.3
vec[c(FALSE, TRUE)] # ресайклінг до FALSE TRUE FALSE TRUE
#> [1] 5.2 1.4

vec[TRUE]
#> [1] 2.1 5.2 2.3 1.4
vec[[TRUE]]
#> [1] 2.1
```

Ніщо повертає оригінальний вектор:

```
vec[]
#> [1] 2.1 5.2 2.3 1.4
```

Нуль повертає вектор довжиною нуль:

```
vec[0]
#> numeric(0)
```

Об'єкт, що подається до [є звичайним вектором, що дозволяє використовувати для сабсетингу значення, що зберігаються у інших векторах, подавати до [функції, якщо дані функції повертають вектор логічного, чисельного чи текстового типу та виконувати у [логічні та арифметичні операції:

```
y <- c(1, 2, 2, 3, 4)
vec[y]
#> [1] 2.1 5.2 5.2 2.3 1.4

vec[2+2]
#> [1] 1.4
```

Як вже було зазначено, можливо використовувати сабсетинг разом із <-

```
vec[[4]] <- 10
vec[[5]] <- 5.5
vec
#> [1]  2.1  5.2  2.3 10.0  5.5
vec[[1]] <- "a"
vec
#> [1] "a"   "5.2" "2.3" "10"  "5.5"
```

`which()` та деякі інші команди

Функція `which(x)` повертає **індекс** елементів вектору, що задовольняють вказану логічну умову:

```r
set.seed(2312)
x <- sample(100, 10)
x
#>  [1] 71 44 95 13 74 41 17 85 10 35

which(x > 10 & x < 35)
#> [1] 4 7
which.max(x)
#> [1] 3
which.min(x)
#> [1] 9
```

Низка інших функцій для роботи з векторами

```r
order(x) # повертає індекси сортованого вектору
#>  [1]  9  4  7 10  6  2  1  5  8  3
sort(x)  # повертає сортований вектор
#>  [1] 10 13 17 35 41 44 71 74 85 95
rev(c(3, 10, 1)) # перегортає вектор
#> [1]  1 10  3
unique(rep(letters[5:9], 10)) # повертає унікальні значення
#> [1] "e" "f" "g" "h" "i"
table(rep(letters[5:9], 10))  # крос-табуляція, докладніше пізніше
#>
#>  e  f  g  h  i
#> 10 10 10 10 10
```

Сабсетинг листів

Майже як вектори, але [завжди повертає структуру типу лист

```r
example_ls <- list(a = "a", b = 1:3, c = list("One", 2))
example_ls[1]
#> $a
#> [1] "a"
example_ls[3]
#> $c
#> $c[[1]]
#> [1] "One"
#>
#> $c[[2]]
#> [1] 2
typeof(example_ls[3])
#> [1] "list"
typeof(example_ls[1])
#> [1] "list"
```

Оператор [[повертає єдиний елемент з відповідним типом

```r
example_ls[[1]]   # референсинг по індексу
#> [1] "a"
example_ls[["b"]] # референсинг по імені
#> [1] 1 2 3
typeof(example_ls[[1]])
#> [1] "character"
typeof(example_ls[["b"]])
#> [1] "integer"
```

```r
example_ls[["c"]]
#> [[1]]
#> [1] "One"
#>
#> [[2]]
#> [1] 2
typeof(example_ls[["c"]])
#> [1] "list"
example_ls[["c"]][[1]]
#> [1] "One"
typeof(example_ls[["c"]][[1]])
#> [1] "character"
```

Оператор `$` фактично є скороченням оператору `[[` для доступу по імені

```r
example_ls$a
#> [1] "a"
example_ls$c[[2]]
#> [1] 2
```

Як і у випадку з атомарним вектором, `<-` може бути використано для заміни або додавання елементів, окрім того `<-` може бути комбіноване з `NULL` для видалення елементу листа

```r
example_ls$c <- NULL       # це видалить елемент
example_ls$w <- list(NULL) # а це буквально додасть елемент зі значенням NULL
```

Матриці

Прямокутні структури, що мають колонки та рядки. Фактично є векторами, що мають атрибут розмірності **dim**. Як і атомарні вектори, матриці можуть містити дані лише одного типу.

Створення матриці командою **matrix()**, дефолтно матриця будується по колонкам

```r
m <- matrix(c(1, 2, 3,
              1, 2, 3,  # вказати кількість колонок або рядків, базове значення 1
              1, 2, 3), nrow = 3)
m
#>      [,1] [,2] [,3]
#> [1,]    1    1    1
#> [2,]    2    2    2
#> [3,]    3    3    3
```

Можливо вказати **byrow=TRUE**, щоб матриця була побудована по рядках

```r
m <- matrix(c(1, 2, 3,
              1, 2, 3,
              1, 2, 3), nrow = 3, byrow = TRUE)
m
#>      [,1] [,2] [,3]
#> [1,]    1    2    3
#> [2,]    1    2    3
#> [3,]    1    2    3
```

Матриці

```r
attributes(m) # матриця 3x3
#> $dim
#> [1] 3 3
dim(m)
#> [1] 3 3
nrow(m) # кількість рядків
#> [1] 3
ncol(m) # кількість колонок
#> [1] 3
```

Звичайний атомарний вектор не має просторової розмірності взагалі

```r
n <- 1:9
dim(n)
#> NULL

dim(n) <- c(1, 9) # тепер має
n
#>      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
#> [1,]    1    2    3    4    5    6    7    8    9

dim(n) <- c(3, 3)
n
#>      [,1] [,2] [,3]
#> [1,]    1    4    7
#> [2,]    2    5    8
#> [3,]    3    6    9
```

Матриці

Так як матриці є векторами у розумінні R звичайні арифметичні та логічні операції дають такий самий результат як з векторами

```
m * n # елемент n[i, j] буде помножено на елемент m[i, j]
#>      [,1] [,2] [,3]
#> [1,]    1    8   21
#> [2,]    2   10   24
#> [3,]    3   12   27
```

Для виконання дій лінійної алгебри є низка своїх операторів, зокрема %*% для множення матриці

```
m %*% n
#>      [,1] [,2] [,3]
#> [1,]   14   32   50
#> [2,]   14   32   50
#> [3,]   14   32   50
```

```
crossprod(X, Y = NULL) # крос-продукт
tcrossprod(X, Y = NULL)
outer(X, Y) # зовнішній продукт
%o%
```

Матриці

Матриця може бути транспонована командою `t(x)`

```
t(n)
#>      [,1] [,2] [,3]
#> [1,]    1    2    3
#> [2,]    4    5    6
#> [3,]    7    8    9
```

Колонки та рядки матриці можуть мати імена, які можуть бути встановлені відповідно командами `colnames()` та `rownames()`, аналогічно `names()` для векторів. Додати нову колонку або рядок до матриці можливо застосуванням `cbind()` та `rbind()`

```
n <- cbind(n, c(0, 0, 0))
rownames(n) <- LETTERS[1:3]
colnames(n) <- letters[1:4]
n
#>   a b c d
#> A 1 4 7 0
#> B 2 5 8 0
#> C 3 6 9 0
```

Сабсетинг матриць

Як вектори, але мають додатковий вимір, тому у [ідуть два значення, перше вказує на рядок, друге на колонку

```r
n[[2, 3]] # другий елемент третьої колонки
#> [1] 8
n[1:2, 1:2]
#>   a b
#> A 1 4
#> B 2 5
n[2, ]  # другий рядок повністю
#> a b c d
#> 2 5 8 0
n[ ,3]  # третя колонка повністю
#> A B C
#> 7 8 9
n[, 3, drop = FALSE] # збереження розмірності
#>   c
#> A 7
#> B 8
#> C 9

typeof(n) # матриця не є типом, як не дивно
#> [1] "double"
class(n)
#> [1] "matrix" "array"
class(n[, 3])
#> [1] "numeric"
class(n[, 3, drop = FALSE])
#> [1] "matrix" "array"
```

P.s Ви могли помітити, що об'єкт окрім класу **`matrix`** також мав клас **`array`** — масив. Окрім двомірних матриць можливо створювати структури вищої розмірності. Сабсетинг такий самий, просто з оглядом на додаткові виміри

```r
arr <- array(1:18, c(3, 3, 2)) # простий приклад, може бути гірше
arr
#> , , 1
#>
#>      [,1] [,2] [,3]
#> [1,]    1    4    7
#> [2,]    2    5    8
#> [3,]    3    6    9
#>
#> , , 2
#>
#>      [,1] [,2] [,3]
#> [1,]   10   13   16
#> [2,]   11   14   17
#> [3,]   12   15   18

arr[1, 3, 2]
#> [1] 16
```

Кадри даних (Data Frames)

Прямокутні структури, що мають колонки та рядки. Подібні за властивостями до листів, фактично кажучи колонки кадрів даних і є векторами листів — таким чином *колонки* мають включати у себе об'єкти одного типу, у той час як *рядки* можуть містити об'єкти різних типів.

Створити кадр даних можливо командою `data.frame()`, вказавши імена та значення для колонок:

```r
df <- data.frame(
  char = c(letters[1:4]),
  int = 1:4,
  dbl = runif(4))
df
#>   char int       dbl
#> 1    a   1 0.2103072
#> 2    b   2 0.2104003
#> 3    c   3 0.4291486
#> 4    d   4 0.1698004
```

Краще зрозуміти як можуть виглядати кадри даних можливо через ознайомлення з класичними датасетами представленими у R, наприклад `mtcars` або `iris`. Або взяти щось менш популярне.

```r
data(crabs, package = "MASS") # це експліцитно завантажить ці данні у Ваш робочій простір
#?MASS::crabs щоб дізнатися більше про набір даних
```

Щоб інтерактивно переглянути кадр даних у більш звичному стилі електронної таблиці викличте `View(x)`

Кадри даних

Швидке знайомство з кадром даних можливо провести переглянувши декілька перших рядків у консолі

```
head(crabs, n = 5)
#>   sp sex index  FL  RW   CL   CW  BD
#> 1  B   M     1 8.1 6.7 16.1 19.0 7.0
#> 2  B   M     2 8.8 7.7 18.1 20.8 7.4
#> 3  B   M     3 9.2 7.8 19.0 22.4 7.7
#> 4  B   M     4 9.6 7.9 20.1 23.1 8.2
#> 5  B   M     5 9.8 8.0 20.3 23.0 8.2
```

Або краще, переглянувши його структуру

```
str(crabs)
#> 'data.frame':    200 obs. of  8 variables:
#>  $ sp   : Factor w/ 2 levels "B","O": 1 1 1 1 1 1 1 1 1 1 ...
#>  $ sex  : Factor w/ 2 levels "F","M": 2 2 2 2 2 2 2 2 2 2 ...
#>  $ index: int  1 2 3 4 5 6 7 8 9 10 ...
#>  $ FL   : num  8.1 8.8 9.2 9.6 9.8 10.8 11.1 11.6 11.8 11.8 ...
#>  $ RW   : num  6.7 7.7 7.8 7.9 8 9 9.9 9.1 9.6 10.5 ...
#>  $ CL   : num  16.1 18.1 19 20.1 20.3 23 23.8 24.5 24.2 25.2 ...
#>  $ CW   : num  19 20.8 22.4 23.1 23 26.5 27.1 28.4 27.8 29.3 ...
#>  $ BD   : num  7 7.4 7.7 8.2 8.2 9.8 9.8 10.4 9.7 10.3 ...
```

Кадри даних

Для приєднання нових колонок / рядків використовуються ті ж команди, що і у випадку матриці

```
colnames(crabs) <- c(colnames(crabs)[1:3], "front_lobe", "rear_wd",
                     "carapace_l", "carapace_wd", "body_depth")
crabs <- cbind(crabs, species = rep("crab", nrow(crabs)))

head(crabs, 5)
#>   sp sex index front_lobe rear_wd carapace_l carapace_wd body_depth species
#> 1  B   M     1        8.1     6.7       16.1        19.0        7.0    crab
#> 2  B   M     2        8.8     7.7       18.1        20.8        7.4    crab
#> 3  B   M     3        9.2     7.8       19.0        22.4        7.7    crab
#> 4  B   M     4        9.6     7.9       20.1        23.1        8.2    crab
#> 5  B   M     5        9.8     8.0       20.3        23.0        8.2    crab
```

Сабсетинг є дещо міксом між матрицями та листами, але зазвичай це не стає проблемою

```
small_crabs <- head(crabs, n = 3) # маленький шматок для демонстрації
```

Сабсетинг кадрів даних

```
small_crabs[4:9]   # колонки з 4 по 9, сабсетинг як лист
#>   front_lobe rear_wd carapace_l carapace_wd body_depth species
#> 1        8.1     6.7       16.1        19.0        7.0    crab
#> 2        8.8     7.7       18.1        20.8        7.4    crab
#> 3        9.2     7.8       19.0        22.4        7.7    crab
small_crabs[, 4:9] # теж саме, сабсетинг як матриця
#>   front_lobe rear_wd carapace_l carapace_wd body_depth species
#> 1        8.1     6.7       16.1        19.0        7.0    crab
#> 2        8.8     7.7       18.1        20.8        7.4    crab
#> 3        9.2     7.8       19.0        22.4        7.7    crab
small_crabs[2, 1:4]
#>   sp sex index front_lobe
#> 2  B   M     2        8.8

str(small_crabs["index"])   # зберігає структуру
#> 'data.frame':    3 obs. of  1 variable:
#>  $ index: int  1 2 3
str(small_crabs[, "index"]) # симліфікація до атомарного вектору, як і у випадку матриці
#>  int [1:3] 1 2 3
str(small_crabs[, "index", drop = FALSE])
#> 'data.frame':    3 obs. of  1 variable:
#>  $ index: int  1 2 3
str(small_crabs$index) # також дроп структури
#>  int [1:3] 1 2 3
```

Сабсетинг кадрів даних

```r
typeof(small_crabs["index"]) # можна впевнитися, що колонки кадрів дійсно є листами
#> [1] "list"

# як і з листами, NULL можна використати, щоб видалити колонку
small_crabs$species <- NULL
small_crabs
#>   sp sex index front_lobe rear_wd carapace_l carapace_wd body_depth
#> 1  B   M     1        8.1     6.7       16.1        19.0        7.0
#> 2  B   M     2        8.8     7.7       18.1        20.8        7.4
#> 3  B   M     3        9.2     7.8       19.0        22.4        7.7

# так само додати нову
small_crabs$new_val <- NA # NA для прикладу, сюди можна присвоїти значення одразу
small_crabs
#>   sp sex index front_lobe rear_wd carapace_l carapace_wd body_depth new_val
#> 1  B   M     1        8.1     6.7       16.1        19.0        7.0      NA
#> 2  B   M     2        8.8     7.7       18.1        20.8        7.4      NA
#> 3  B   M     3        9.2     7.8       19.0        22.4        7.7      NA

# базове практичне застосування з іншими функціями
mean(small_crabs$rear_wd)
#> [1] 7.4
max(small_crabs$front_lobe)
#> [1] 9.2
```

`factor` та деякі інші S3-вектори

Вектори можуть мата атрибут `class`, що перетворює їх на S3-об'єкти, при передачі таких об'єктів до generic-функцій для їх обробки буде викликано метод специфічний конкретному класу.

Фактори — `factor` — є репрезентацією категоріального типу даних з певним **фіксованим** набором рівнів. Кожен рівень фактору є кодованим цілочисловим значенням. Якщо знову переглянути структуру датасету `crabs` то можна побачити, що перша та друга колонка значаться як `Factor w/ 2 levels` з числовими значеннями.

Колонка `crabs$sex`, що містить дані про стать особини, є *категоріальним* типом даних з двома можливими рівнями F та M, що відповідно кодовані числами 1 та 2 у структурі кадру даних

```
levels(crabs$sex)
#> [1] "F" "M"
class(crabs$sex)
#> [1] "factor"
```

"Здерти" атрибут класу можливо функцією `unclass(x)`, що може дозволити більш явно побачити, що структура фактору є надбудовою над вектором типу `integer`

```
typeof(crabs$sex)
#> [1] "integer"
```

`factor` та деякі інші S3-вектори

Фактори можуть бути конвертовані до звичайного вектору типу `character` та навпаки. Фактор може бути створено відповідною командою `factor()`

```
crustacea <- factor(c("crab", "shrimp", "shrimp"))
crustacea
#> [1] crab   shrimp shrimp
#> Levels: crab shrimp

crustacea <- factor(c("crab", "shrimp", "shrimp"), levels = c("crab", "shrimp", "krill"))
crustacea
#> [1] crab   shrimp shrimp
#> Levels: crab shrimp krill
```

Команда `table()`, що повертає об'єкт відповідного класу `table`, використовує перехресну класифікацію факторів для побудови **таблиці спряженості** (contingency table) — таблиці, яка відображає багатофакторний частотний розподіл змінної

```
table(crustacea)
#> crustacea
#>   crab shrimp  krill
#>      1      2      0
```

`factor` та деякі інші S3-вектори

Працюючи з даними також можливо зустрітися з класами векторів що є репрезентацією часу:

- `Date` — репрезентація дати
- `POSIXct` та `POSIXlt` — репрезентація дати-часу
- `difftime` — репрезентація проміжку часу

```r
Sys.Date()
#> [1] "2024-03-03"
class(Sys.Date())
#> [1] "Date"
Sys.time()
#> [1] "2024-03-03 15:43:21 EET"
class(Sys.time())
#> [1] "POSIXct" "POSIXt"

difftime(as.Date("2024-03-04"), as.Date("2024-03-01"))
#> Time difference of 3 days
difftime(as.Date("2024-03-04"), as.Date("2024-03-01"), units = "mins")
#> Time difference of 4320 mins
```

Усі ці класи є надбудовою над вектором типу `double`

```r
typeof(as.Date("2024-03-01"))
#> [1] "double"
```

*POSIX означає Portable Operating System Interface, ct - calendar time та lt - local time

Написання функцій

Функція створюється відповідною командою `function(...)` і складається з трьох обов'язкових компонентів — аргументів, тіла та середовища виконання, інформацію про які можна отримати командами `formals()`, `body()` та `environment()` відповідно. Функції зв'язуються зі своїм іменем як і будь-який інший об'єкт

```r
fun <- function(a, b, c, x) { # аргументи
  a*x^2 + b*x + c             # тіло функції
}                             # середовище залежить від того, де функція створена
```

Імпліцитно функція повертає останній об'єкт, який було отримано у ході виконання коду. Експліцитно вказати які об'єкти мають бути поверненні можливо викликом `return()`

```r
fun2 <- function(x, y) {
  x^2
  return(y)
}

fun2(5, 1)
#> [1] 1
```

Написання функцій

Прив'язка до імені не обов'язкова, функція може бути анонімною, зазвичай анонімні функції використовуються у середині команд сімейства `apply()` або їх аналогів

```r
(function (x) 3^2)()
#> [1] 9
sapply(1:10, function(x) x^2)
#>  [1]   1   4   9  16  25  36  49  64  81 100
sapply(1:10, \(x) x^2) # скорочення для анонімної функції
#>  [1]   1   4   9  16  25  36  49  64  81 100
```

Функції також можливо організувати у лист:

```r
funs <- list(
  square = function(x) x^2,
  cube = function(x) x^3,
  tesseract = function(x) x^4
)

funs$square(2)
#> [1] 4
funs$cube(2)
#> [1] 8
funs$tesseract(2)
#> [1] 16
```

Lexical scoping та Ліниві обчислення (Lazy evaluation)

У R аргументи функцій обчислюються тільки у випадку безпосереднього звернення до них

```r
hello <- function(x) {
  print("Hello")
}

exists("somebody") # об'єкт з такою назвою наразі не існує у середовищі
#> [1] FALSE
hello(somebody)    # але функція все одно виконується
#> [1] "Hello"
```

Модифіковане завдання із SICP (в оригіналі спрямовано на демонстрацію різниці між normal-order та applicative-order evaluation)

```r
p <- function() {
  p()
}

test_xy <- function(x, y) {
  if (x == 0) {
    0
  } else {
    y
  }
}

test_xy(0, p())
#> [1] 0
```

Lexical scoping та Ліниві обчислення (Lazy evaluation)

Лексичний скоупінг — пошук значень асоційованих з певними іменами, слідує у R таким правилам:

- Name masking — імена визначені усередині функції маскують імена визначені поза функцією
- Functions vs variables — так як функції є звичайними об'єктами, правило маскування імен застосовується і до функцій
- Fresh start — при кожному виклику функції для її виконання створюється нове середовище, тому кожен виклик є незалежним. Функція не зберігає інформації про значення змінних, що були отримані у минулому виклику
- Dynamic lookup — результат кожного нового виклику функції може відрізнятися і залежати від стану об'єктів, що знаходяться поза середовищем виконання функції

Приклад маскування імені, за іншими прикладами до Advanced R 2ed, Chapter 6.4

```r
x <- 10
y <- 20
fun <- function() {
  x <- "Some string"
  c(x, y)
}

fun()
#> [1] "Some string" "20"
c(x, y)
#> [1] 10 20
```

Control flow

Докладніше `help("Control")`. У R присутні два типи операторів для контролю потоку

- оператори вибору — `if` / `else` та `switch()`
- оператори циклу — `for`, `while` та `repeat`

А також допоміжні оператори для контролю усередині циклу — `next` та `break`.

```r
if (condition) do_something
if (condition) do_something else do_something_else

for (item in sequence) do_action
while (condition == TRUE) do_action
repeat action

for (item in sequence) {
  if (condition)
    next
  do_action
}
repeat {
  if (condition)
    break
  action
}
```

У R також є низка команд для зворотного діагностичного фідбеку до користувача — `stop()`, `message()` та `warnings()`

Control flow

Приклад з `if` / `else`

```r
is_even <- function(x) {
  if (x == 0) {
    cat("Not odd, not even:", x, "is zero")
  } else if (x %% 2 != 0) {
    cat("No", x, "is an odd number")
  } else {
    cat("Yes", x, "is an even number")
  }
}
```

Простий приклад `for` ітерації

```r
for (i in letters[1:5]) print(i)
#> [1] "a"
#> [1] "b"
#> [1] "c"
#> [1] "d"
#> [1] "e"
for (i in seq_along(letters[1:5])) print(letters[[i]]) # більш надійний варіант
#> [1] "a"
#> [1] "b"
#> [1] "c"
#> [1] "d"
#> [1] "e"
```

Control flow

Приклад `switch()` — альтернатива `if` / `else` для окремих випадків

```r
grade <- function(g) {
  switch(g,
      A = "> 89%",
      B = "75 - 89%",
      C = "60 - 74%",
    `F` = "< 60%",
      stop("Invalid value")
  )
}
grade("B")
#> [1] "75 - 89%"
```

Варіант якщо декілька вхідних опцій дають однаковий вихід

```r
grade_pass <- function(g) {
  switch(g,
      A = ,
      B = ,
      C = "passed",
    `F` = "failed",
      stop("Invalid value")
  )
}
grade_pass("B")
#> [1] "passed"
grade_pass("F")
#> [1] "failed"
```

Control flow

Функція з `for` циклом

```r
for_petri <- function(n) {
  for (i in 1:n) {
    cat("There", if (n == 1) "was" else "were", n, "Petri", if (n==1) "dish." else "dishes.",
    "One fell down. There are", n - 1, "Petri dishes left.\n")
    n <- n-1
  }
}
for_petri(5)
#> There were 5 Petri dishes. One fell down. There are 4 Petri dishes left.
#> There were 4 Petri dishes. One fell down. There are 3 Petri dishes left.
#> There were 3 Petri dishes. One fell down. There are 2 Petri dishes left.
#> There were 2 Petri dishes. One fell down. There are 1 Petri dishes left.
#> There was 1 Petri dish. One fell down. There are 0 Petri dishes left.
```

Та ж сама функція, але з `while` циклом та векторизованим `ifelse()`

```r
while_petri <- function(n) {
  while (n > 0) {
    cat("There", ifelse(n == 1, "was", "were"), n, "Petri", ifelse(n == 1, "dish.", "dishes.",
        "One fell down. There are", n - 1, "Petri dishes left.\n")
    n <- n-1
  }
}
```

*`cat()` виводить об'єкти, об'єднуючи їх репрезентацію, альтернативно тут можна використати `print(paste())` або той же `print(c())`

Векторизований `ifelse()`

As it says. Бере вхідний вектор значень і повертає вихідний вектор значення якого залежать від задовільненя умови — `ifelse(test_condition, yes, no)`

```r
x <- 1:10
ifelse(x %% 2 == 0, "even", "odd")
#>  [1] "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even"
```

Здирає атрибути класу, тому вихідні результати інколи можуть бути непередбачуваними. Приклад прямо з документації функції

```r
x <- seq(as.Date("2000-02-29"), as.Date("2004-10-04"), by = "1 month")
head(x)
#> [1] "2000-02-29" "2000-03-29" "2000-04-29" "2000-05-29" "2000-06-29"
#> [6] "2000-07-29"
y <- ifelse(as.POSIXlt(x)$mday == 29, x, NA)
head(y)
#> [1] 11016 11045 11076 11106 11137 11167
class(y) <- "Date"
head(y)
#> [1] "2000-02-29" "2000-03-29" "2000-04-29" "2000-05-29" "2000-06-29"
#> [6] "2000-07-29"
```

*Пакет dplyr, про який ми будемо говорити у наступний раз, має більш стабільну версію функції — `is_else()`

Сімейство `apply()`

Картує функцію на кожен елемент масиву або листу. Як факт, ці функції є прихованим `for`-циклом. **Коректно** написаний `for`-цикл буде займати приблизно стільки ж часу на виконання скільки функція сімейства `apply()`.

Таблиця `apply` функцій

функція	вхід	вихід
`apply`	матриця або масив	вектор, масив або лист
`lapply`	лист або вектор	лист
`sapply`, `vaply`	лист або вектор	вектор, матриця, масив
`tapply`	дані розбиті на групи за комбінацією рівнів факторів	вектор або лист
`mapply`	листи або вектори	лист, вектор або масив
`rapply`	лист	вектор або лист
`eapply`	середовище	лист

Сімейство `apply()`

У більшості випадків Вам знадобиться лише `lapply()`, у прикладі нижче насправді можливо використати `apply(crabs[4:8], 2, sd)`, але функція `lapply()` є швидшою

```
rbind(lapply(crabs[4:8], mean)) # для цього насправді є векторизований colMeans()
#>       FL     RW      CL      CW      BD
#> [1,] 15.583 12.7385 32.1055 36.4145 14.0305
rbind(lapply(crabs[4:8], sd))
#>       FL       RW      CL       CW       BD
#> [1,] 3.495325 2.57334 7.118983 7.871955 3.424772
```

Інколи для того, щоб швидко перевірити певний групований статистичний підсумок (e.g. середнє певного параметру для кожного рівня конкретного фактору) зручним буває `tapply()`

```
tapply(crabs$FL, crabs$sex, mean)
#>      F      M
#> 15.432 15.734
```

Також існує функція `aggregate()`

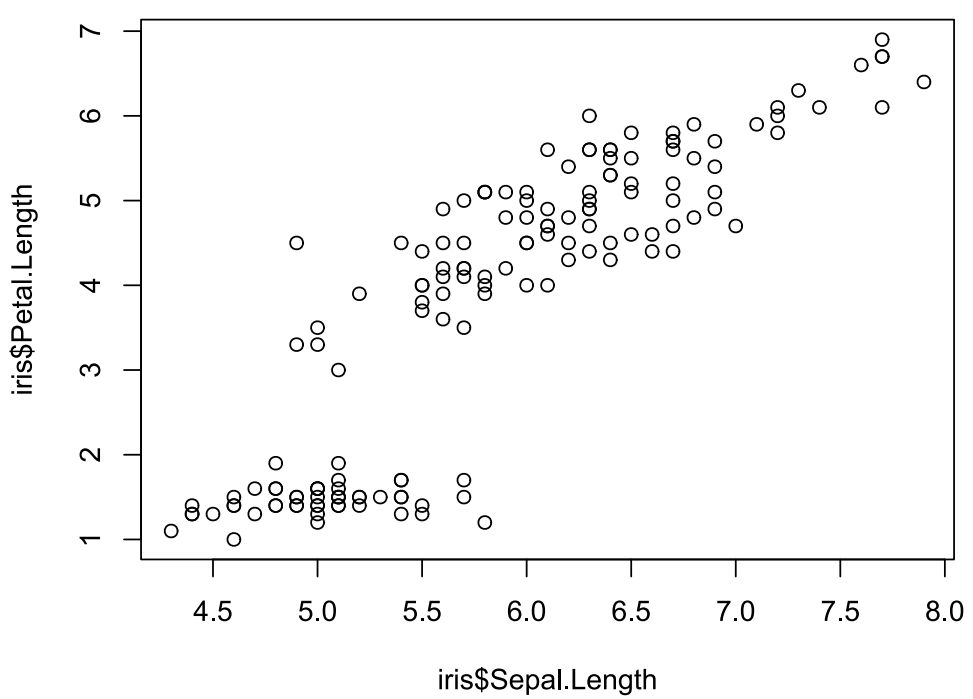
```
aggregate(FL ~ sex, crabs, mean)
#>   sex     FL
#> 1   F 15.432
#> 2   M 15.734
aggregate(. ~ sex, crabs, mean)
#>   sex  sp index     FL     RW     CL     CW     BD
#> 1   F 1.5  25.5 15.432 13.487 31.360 35.830 13.724
#> 2   M 1.5  25.5 15.734 11.990 32.851 36.999 14.337
```

Базова графіка

Базові графічні можливості R насправді є досить широкими, проте (на мою думку) граматика стандартних графічних бібліотек сильно поступається у інтуїтивності граматиці популярного ggplot2

Тим не менш, найпростіші графіки можливо швидко створити за допомогою низки простих коротких команд, що є ідеальним для процесу Explaratory Data Analisys. Команда `plot(x, y)` за умовчанням створює класичну діаграму розсіяння (scatterplot)

```
1  plot(iris$Sepal.Length, iris$Petal.Length)
```



Деякі інші команди для виклику графіки

- `hist`
- `pairs`
- `boxplot`
- `barplot`
- `dotchart`
- `mosaicplot` та `spineplot`
- `image`, `contour`, `heatmap`

Також команда `par(...)` для додаткового налаштування параметрів базової графіки

Базове використання статистичних функцій

Більшість як нативних так і імпортованих функцій працюють з кадрами даних або векторами та використовують синтаксис типу `fun(y ~ x, data, ...)` або `fun(y, x)`, де `y` та `x` є залежною та незалежною змінною (або змінними) відповідно

```r
data("PlantGrowth")
kruskal.test(weight ~ group, PlantGrowth)
#>
#>  Kruskal-Wallis rank sum test
#>
#> data:  weight by group
#> Kruskal-Wallis chi-squared = 7.9882, df = 2, p-value = 0.01842

pairwise.t.test(PlantGrowth$weight, PlantGrowth$group, p.adjust.method = "bonf")
#>
#>  Pairwise comparisons using t tests with pooled SD
#>
#> data:  PlantGrowth$weight and PlantGrowth$group
#>
#>      ctrl  trt1
#> trt1 0.583 -
#> trt2 0.263 0.013
#>
#> P value adjustment method: bonferroni
```

The Pipe

При виконанні ланцюжку команд одну функцію можливо помістити у середину іншої, тому

```r
crabs_male <- subset(crabs, sex == "M")
aggregate(. ~ sp, crabs_male, mean)
```

є еквівалентом

```r
aggregate(. ~ sp, subset(crabs, sex == "M"), mean)
```

Нативний інфікс оператор `|>` (R 4.1+) або `%>%` з пакету `magrittr` дозволяють створювати ланцюг команд подібного типу, що (інколи) робить їх більш читабельними.

```r
crabs |> subset(sex == "M") |> aggregate(. ~ sp, mean)
#>   sp sex index    FL    RW    CL    CW    BD
#> 1  B   2  25.5 14.842 11.718 32.014 36.810 13.350
#> 2  O   2  25.5 16.626 12.262 33.688 37.188 15.324

library(magrittr)
crabs %>% subset(sex == "M") %>% aggregate(. ~ sp, mean)
#>   sp sex index    FL    RW    CL    CW    BD
#> 1  B   2  25.5 14.842 11.718 32.014 36.810 13.350
#> 2  O   2  25.5 16.626 12.262 33.688 37.188 15.324
```

Про різницю між нативним pipe-оператором та оператором magrittr можна прочитати тут. Важливо зазначити, що даний оператор першочергово розроблявся для роботи з бібліотеками екосистеми `tidyverse` і його нативна версія є саме *нововведенням* для базового R

P.s. Не ускладнюйте собі життя

Якщо варіанти програмних конструкцій типу

```r
y <- h(g(f(x)))
```

та її ріре-версія

```r
y <- x |> f() |> g() |> h()
```

виглядають однаково заплутано та контрпродуктивно, то краще створіть тимчасові проміжні змінні

```r
a <- f(x)
b <- g(a)
y <- h(b)
```

Якщо Вам складно зрозуміти, що відбувається при виклику функції сімейства `apply` або інших функціоналів, краще напишіть конструкцію з `for`-циклом