base R, pt.1

https://github.com/llevenets/intro-r

Інтро

- R це мова програмування для статистичних обчислень та візуалізації даних яка використовується вже понад 30 років - перші бінарні файли були опубліковані у серпні 1993 року. Створений Россом Іхакою та Робертом Джентльменом, на основі мови S та Sheme, R був широко прийнятий на використання у області дата-майнінгу, аналізу даних та біоінформатики
- R є функціональною програмною мовою і створювався специфічно для статистичних обчислень, через що має низку особливостей, наприклад індексація починаючи з одиниці, векторизація обчислень та lazy evaluation
- Ядром R є інтерпретована комп'ютерна мова, яка дозволяє розгалуження та цикли, а також модульне програмування за допомогою функцій. Більшість видимих для користувача функцій у R написані на R, проте частина базових функцій є написаною на C, C++ та FORTRAN
- Програмне забезпечення R є вільним програмним забезпеченням із відкритим кодом, офіційною частиною GNU project і розповсюджується під копілефт ліцензією GNU GPL
- R підтримується сімейством Unix-подібних ОС, Windows та Mac ОС
- R поставляється зі своїм власним вбудованим CLI
- RStudio Desktop є вільним (ліцензія GNU AGPL) кросплатформеним IDE початково розробленим специфічно під R (на сьогодні розширеним до роботи з низкою інших популярних мов).
- З R також достатньо зручно працювати використовуючи популярний у науковій спільноті відкритий інтерактивний блокнот Jupyter Notebook/Lab або використовуючи текстовий редактор Emacs з аддоном ESS

Рідлист

Як завжди одним з перших місць для пошуку інформації є офіційні мануали:

- R FAQ — Frequently Asked Questions on R
- An Introduction to R

А також:

- Advanced R, Second Edition (Hadley Wickham, 2019) — я дуже раджу спробувати прочитати розділ Foundations, або хоча б глави з другої (Names and values) по четверту (Subsetting)

Усяке різне:

- What every computer scientist should know about floating-point arithmetic (David Goldberg, 1991) — лонгрід про особливості арифметичних операції з плаваючою комою
- Structure and Interpretation of Computer Programs, second edition (Harold Abelson, Gerald Jay Sussman and Julie Sussman, 1996) — якщо Ви зацікавтеся програмуванням, то завжди можете спробувати прочитати оцей класичний трактат на 800 сторінок задля інтересу

Бібліотеки

Функція є серцем програмної мови. Функція це набір команд, чітко визначена поведінка, упорядкований самостійний блок коду, що виконує конкретне завдання.

Функції (а також інші об'єкти) можуть бути зібрані у бібліотеки або пакети. R поставляється з набором системних бібліотек (System Library), що включає у себе пакет базових функції, формальних методів та класів (`base`, `methods`), пакети для розробки бібліотек та аналізу коду, пакети графіки (`graphics`, `lattice` etc), низку пакетів для статистичного аналізу та моделювання (`stats`, `nlme`, `survival` etc), а також пакет наборів даних (`datasets`)

Низка бібліотек необхідних для базового функціювання R автоматично завантажуються при запуску нового робочого сеансу. Додаткові бібліотеки можуть бути завантажені (приєднанні) у робочий простір командою `library()`

```
1  library(MASS)
```

Викликати індивідуальну функцію або інший об'єкт без завантаження усієї бібліотеки можливо використавши конструкцію типу `package::function()`

```
1  # функція пошуку раціонального наближення методом неперервного дробу з пакету MASS
2  MASS::fractions(sqrt(2))
3  #> [1] 8119/5741
```

Подібний синтаксис також може бути застосований у випадку коли одночасно завантажені бібліотеки мають однакове ім'я для якоїсь функції і одна з них маскує іншу

Щоб завантажити бібліотеку під час робочої сесії вона має бути встановлена на ваш локальний комп'ютер, чого можна досягти командою `install.packages()` або скориставшись вкладкою Packages GUI RStudio

Основним репозиторієм бібліотек R слугує CRAN, що на даний момент хостить понад 20к бібліотек

```r
# Бібліотека, що фактично складається лише з датасету пінгвінів Палмера
install.packages("palmerpenguins")
```

Репозиторієм бібліотек, що є розробленими специфічно для цілей біоінформатиків слугує FOSS-проект Bioconductor. Bioconductor має свій власний менеджер пакетів, що має бути попередньо встановлений як звичайна бібліотека

```r
# DO NOT RUN
install.packages("BiocManager")
BiocManager::install("palmerpenguins") # лише для прикладу, Biocondctor не хостить даний паке
```

Бібліотека також може бути встановлена безпосередньо із сурса

```r
# необхідно вказати шлях до директорії куди збережено архів бібліотеки
install.packages("./palmerpenguins_0.1.1.tar.gz", type = "source", repos = NULL)
```

Dev-версія бібліотеки за необхідності зазвичай може бути встановлена з GitHub розробника

```r
# install.packages("remotes")
remotes::install_github("allisonhorst/palmerpenguins")
```

Усі бібліотеки поставляються з розгорнутою документацією, прикладами та віньєтками. Розгорнута інформація про бібліотеку - опис, версія, автор, сурс-архів та бінарні файли, залежності, імпорти, документація у форматі .pdf та короткі туторіали зазвичай можуть бути знайдені на відповідній сторінці CRAN або Bioconductor, e.g. palmerpenguins: Palmer Archipelago (Antarctica) Penguin Data

Для пошуку документації також можуть бути використані сайти-агрегатори типу RDocumentation та rdrr

За документацією (та власне кодом!) також завжди можна звернутися безпосередньо до репозиторію розробників, зазвичай на GitHub. Популярні бібліотеки або сімейства бібліотек нерідко мають свої власні сайти з документацією, e.g. palmerpenguins 0.1.1

Звернутися до документації під час роботи R можливо за допомогою команди `help()` або `?`

```r
# виклик допомоги для функції виклику допомоги

help("help") # " help is the primary interface to the help systems. "
?help

help("?") # " Documentation Shortcunts - These functions provide access to documentation."
?`?`      # Зверніть увагу на ` ` навколо функції

help("??") # " Search the Help System - Allows for searching the help system for documentati
?`??`      # matching a given character string in the (file) name, alias, title, concept... "
```

`c()` та `<-`

Оператор призначення (assignOps) `<-` та оператор конкатенації `c()` виконують дві прості функції – пов'язують об'єкт (значення, змінну) із іменем та комбінують об'єкти у вектор (або лист) відповідно

```r
x <- 1
print(x) # виклик print() не є обов'язковим
#> [1] 1

y <- c(1, 2)
print(y)
#> [1] 1 2

z <- c(x, y)
print(z) # значення ідентичне print(c(x, y))
#> [1] 1 1 2
```

Викликавши `help("assignOps")` можна виявити існування двох інших операторів:

- Оператор суперпризначення `<<-`, що може використовуватися у функціях для зміни глобальних змінних;
- Оператор `=`, що є подібним (хоча і не ідентичним) `<-`, але типово використовується для зв'язування іменованих аргументів функції зі значеннями;
- А також дзеркальні `->` та `->>`, що виконують призначення зліва направо

Стандартні `<-`, `<<-` та `=` виконують призначення справа наліво

```r
a <- b <- c <- 6
print(a)
#> [1] 6

print(c(a, b, c))
#> [1] 6 6 6
```

`<-` не “створює об’єкт x зі значенням n”, натомість `<-`:

- Створює об’єкт - вектор, що містить у собі певні значення n
- Прив’язує цей об’єкт до імені x

З цим пов’язані деякі неочевидні властивості алокації пам’яті під об’єкти

```r
# install.packages("lobstr")
x <- c(1, 2, 4)
lobstr::obj_addr(x)
#> [1] "0x2032838a978"

y <- x
lobstr::obj_addr(y)
#> [1] "0x2032838a978"
```

y не містить у собі копію x, два імені є прив’язаними до одного й того ж самого об’єкту у фізичній пам’яті комп’ютера

```r
x[[3]] <- 3
lobstr::obj_addr(x) # нова адреса
#> [1] "0x2032885c2e8"

lobstr::obj_addr(y) # продовжує вказувати на стару адресу
#> [1] "0x2032838a978"

print(x) # змінився
#> [1] 1 2 3
print(y) # все ще містить первісне значення
#> [1] 1 2 4
```

При спробі модифікації об'єкта пов'язаного з іменем `x` було створено його копію, змінено, після чого ім'я `x` було прив'язано до новоствореної копії об'єкту — це є copy-on-modify behaviour. Ця поведінка є характерною для абсолютної більшості об'єктів R.

Виключеннями є середовища та об'єкти з єдиною прив'язкою які виявляють modify-in-place bahaviour (що можна протестувати при взаємодії з нативним CLI R, результати отримані при інтерактивній взаємодії з консолью RStudio не відповідають дійсності через технічні особливості відеозображення об'єктів у панелі середовища)

Дізнатися чи відбувається копіювання певного об'єкту при виконанні якоїсь дії можна шляхом виклику `tracemem(x)`, де `x` є відповідним об'єктом інтересу.

Докладніше про різницю між об'єктом та його іменем, copy-on-modify та modify-in-place поведінкою для векторів та інших структур можна прочитати у другому розділі Advanced R, "Names and values"

Синтактичні імена

Об'єкти є зв'язаними з іменами і дані імена мають слідувати певним правилам, ім'я об'єкту у R може включати у себе літери, цифри, знак `.` та `_` , але не може починатися із цифри або `_` . Імена також не можуть дублювати т.з. зарезервовані слова, повний список яких можна переглянути викликавши `help("Reserved")`

Що вважається за "літери" насправді визначається вашою локальною машиною, що дозволяє мені зробити наприклад так:

```
"єґї" <- "hi from Ukraine"
print(єґї)
#> [1] "hi from Ukraine"
```

На практиці, аби уникнути проблем з передачею даних між комп'ютерами варто обмежитися використанням літер латиниці AZaz, що кодуються ASCII

Обійти правила синтактичного іменування можливо використавши зворотні лапки :

```
`_bad name` <- "Just don't do things like this"
print(`_bad name`)
#> [1] "Just don't do things like this"
`if` <- "Don't do things like that either"
print(`if`)
#> [1] "Don't do things like that either"
```

Ви майже гарантовано зустрінетесь з не-синтактичними іменами при обробці даних

Арифметичні оператори

R має стандартний набір операторів для виконання арифметичних дій над чисельними або комплексними векторами або об'єктами, що можуть бути перетвореними на них. За деталями щодо точності виконуваних операції та подібним до `help("Arithmetic")`

```r
x + y
x - y
x * y
x / y
x ^ y   # можливо також використання синтаксису типу x ** y
x %% y  # modulo operation, повертає залишок від ділення
x %/% y # повертає цілу частину від ділення, округлення у меншу сторону
```

Найбільш класичний приклад використання modulo operation, визначення чи є число парним

```r
is_even <- function(x) {
  if (x %% 2 != 0) {
    cat("No,", x, "is an odd number")
  } else {
    cat("Yes,", x, "is an even number")
  }
}

is_even(5)
#> No, 5 is an odd number

is_even(2)
#> Yes, 2 is an even number
```

Базові математичні функції

```r
sqrt(x) # квадратний корінь
abs(x)  # абсолютне значення

cos(x) sin(x) tan(x) # та інші тригонометричні функції

sum(...) prod(...) # сума та продукт
min(...) max(...)  # мінімальне та максимальне значення

cumsum(x) cumprod(x) # кумулятивна сума та продукт
cummin(x) cummax(x)  # мініма та максима

log(x, base = exp(1)) log10() log2() # логарифми

round(x, digits = n) signif(x, digits = n) # округлення значень
trunc(x) floor(x) ceiling(x)
```

```r
sum(1, 5, 10) # ... символізує, що ця функція може приймати довільну кількість аргументів
#> [1] 16
cumsum(c(1, 5, 10)) # ця функція приймає лише один аргумент, зверніть увагу на використання c
#> [1]  1  6 16

# різниця між round() та signif()
round(0.005765, 3)
#> [1] 0.006
signif(0.005765, 3) # зверніть увагу що 5 округлюється у меншу сторону
#> [1] 0.00576
```

Бінарні логічні та реляційні оператори

Логічні оператори — `help("Logical")`

```r
!x      # NOT, логічна негація

x & y   # AND, кон'юнкція
x && y

x | y   # OR, диз'юнкція
x || y

xor(x, y) # elementwise exclusive OR, виняткова диз'юнкція

isTRUE(x)
isFALSE(x)
```

Для роботи з даними використовуються **&** та **|**, що виконують обчислення зліва направо, над кожним елементом, доки не буде визначено результат. Оператори **&&** та **||** виконують дії виключно з векторами довжиною один, повертають одне логічне значення та обчислюють по short-circuit, тому використовуються для написання програмних конструкцій

```r
!(TRUE & FALSE) # правило де Моргана
#> [1] TRUE
!(TRUE | FALSE)
#> [1] FALSE

TRUE & (FALSE | TRUE) # дистрибутивність операцій
#> [1] TRUE
TRUE | (FALSE & TRUE)
#> [1] TRUE
```

Бінарні логічні та реляційні оператори

Реляційні оператори — `help("Comparison")`

```r
x < y
x > y
x <= y
x >= y
x == y # еквівалентні
x != y # не еквівалентні
```

Функції `any(...)` та `all(...)` перевіряють чи є хоча б одне значення істинними та чи є усі значення істинними відповідно, завжди повертають логічний вектор довжиною 1

```r
c(0, 2, 2, 1) > 0
#> [1] FALSE  TRUE  TRUE  TRUE

any(c(0, 2, 2, 1) > 0)
#> [1] TRUE
all(c(0, 2, 2, 1) > 0)
#> [1] FALSE
```

Функція `identical(x, y)` є способом перевірки чи є два значення ідентичні, буквально:

```r
1 == 1L          # 1.00 дійсно є еквівалентним 1, але...
#> [1] TRUE
identical(1, 1L) # це два різних типи чисельних значень - double та integer
#> [1] FALSE
```

Оператори сетів

```
help("sets")
```

```
union(x, y)      # союз
intersect(x, y) # перетин
setdiff(x, y)    # різниця (асиметрична)
setequal(x, y)  # чи є сети рівнозначними

is.element(el, set) # чи належить елемент сету, ідентично команді x %in% y
```

```
x <- 1:6
y <- 4:10

union(x, y)
#>  [1]  1  2  3  4  5  6  7  8  9 10
intersect(x, y)
#> [1] 4 5 6
setdiff(x, y)
#> [1] 1 2 3
setequal(x, y)
#> [1] FALSE
is.element(c(1, 4, 10), x)
#> [1]  TRUE  TRUE FALSE
"a" %in% y
#> [1] FALSE

# можливо менш очевидний приклад рівнозначності сетів
setequal(c(1, 1, 0), c(TRUE, TRUE, FALSE))
#> [1] TRUE
```

Bevare of float

```r
x <- 2
y <- 2
x == y
#> [1] TRUE

y <- sqrt(y)^2
y
#> [1] 2

x == y # ???
#> [1] FALSE
```

TL;DR Стиснення нескінченної кількості дійсних чисел у кінцеву кількість бітів вимагає наближеного представлення[пряма цитата] Єдиними числами, що є точно представленими у числовому типі R, є цілі числа та дроби, знаменник яких є степенем 2, усі інші типово округлюються до ≈ 53 знаків після коми, що неминуче веде до помилки округлення

```r
print(y, digits = 20)
#> [1] 2.0000000000000004441
```

У так випадках для порівняння чисельних значень варто використовувати `all.equal(target, current, ...)`, що має певний рівень толерантності до помилки

```r
all.equal(x, y) # забігаючи наперед, також можливо використовувати dplyr::near()
#> [1] TRUE
```

Якщо 53 знаки після коми не достатньо, то арифметичні операції арбітарної точності над числами з плаваючою комою можуть бути виконані за допомогою пакета Rmpfr

Деякі статистичні функції

Базовий пакет `stats`, раджу окремо переглянути індекс документації цього пакету, щоб побачити що ще є

```r
mean(x) sd(x)  # центральна тенденція - середнє та стандартне відхилення
mean(x) mad(x) # центральна тенденція - медіана та медіана абсолютних відхилень

quantile(x, probs) # квантиль, дефолтно повертає 0% 25% 50% 75% 100%
fivenum(x) # п'ятизначний підсумок Тьюкі
IQR(x)     # інтерквантильний розмах
range(...) # мінімум та максимум

cor(x, y) cov(x, y) var(x) # кореляція, коваріація та варіація
```

Також доступними є низка статистичних тестів та моделей, зокрема:

```r
t.test(x, ...) wilcox.test(x, ...) # Критерій Стьюдента та Уілкоксона/Мєнна-Уітні
pairwise.t.test(x, g) pairwise.wilcox.test(x, g)

p.adjust(p, method) # корекція p-значень при багатократних порівняннях
TukeyHSD(x)         # корекція p-значень методом Тьюкі при парних порівняннях

kruskal.test(x, g, ...) # Критерій Краскал-Уолліса
aov(formula, data)      # дисперсійний аналіз, тип I

lm(formula, data)  # лінійна модель
glm(formula, data) # генералізована лінійна модель
```

Build-in Constants

Окрім наборів даних, що йдуть з пакетом `datasets`, а також кожною другою бібліотекою R має:

вбудований алфавіт латиниці

```
letters
#>  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
#> [20] "t" "u" "v" "w" "x" "y" "z"
LETTERS
#>  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
#> [20] "T" "U" "V" "W" "X" "Y" "Z"
```

назви місяців

```
month.abb
#>  [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
month.name
#>  [1] "January"   "February"  "March"     "April"     "May"       "June"
#>  [7] "July"      "August"    "September" "October"   "November"  "December"
```

та число π

```
pi
#> [1] 3.141593
```

Генерація регулярних сиквенсів

Найпростіший метод отримати чисельну послідовність від і до j є використання `:` (двокрапки)

```
1:10
#>  [1]  1  2  3  4  5  6  7  8  9 10
10:1
#>  [1] 10  9  8  7  6  5  4  3  2  1
-(1:10)
#>  [1]  -1  -2  -3  -4  -5  -6  -7  -8  -9 -10
```

Послідовність від і до j з кроком у n можна отримати командою `seq(...)`

```
seq(from = 1, to = 10, by = 1) # теж що і 1:10
#>  [1]  1  2  3  4  5  6  7  8  9 10
seq(0, 12, 4) # від 0 до 12 з кроком 4
#> [1]  0  4  8 12
seq(1, 10, length.out = 5) # відома бажана довжина, автоматично визначений крок
#> [1]  1.00  3.25  5.50  7.75 10.00
```

Послідовність із повторюваних елементів можна отримати командою `rep(x, ...)`

```
rep(10, 3)
#> [1] 10 10 10
rep(c("a", "b", "c"), times = 3) # усю послідовність n разів
#> [1] "a" "b" "c" "a" "b" "c" "a" "b" "c"
rep(c("a", "b", "c"), each = 3)  # кожен елемент послідовності n разів
#> [1] "a" "a" "a" "b" "b" "b" "c" "c" "c"
```

Статистичні розподіли та випадкові числа

Усі таблиці розподілів у пакеті `stats` можна переглянути викликавши `help("Distributions")`, у залежності від префіксу при виклику назви функції розподілу буде повернено:

- `dxxxx` — щільність розподілу, перший аргумент квантиль *x*
- `pxxxx` — кумулятивну щільність розподілу, перший аргумент квантиль *q*
- `qxxxx` — квантиль функції, перший аргумент вірогідність *p*
- `rxxxx` — симуляцію функції, перший аргумент вибірка *n*

Кожна функція також має свої специфічні параметри. Наприклад функція нормального розподілу $N(\mu,\sigma^2)$ має параметр математичного очікування μ та дисперсії випадкової величини σ^2. Функція нормального розподілу R `*norm()` має дефолтні значення `norm(mean=0, sd=1)`, що відповідають **стандартному нормальному розподілу** $N(0, 1)$

```r
dnorm(c(-1, 0, 1))        # Pr(x = a)
#> [1] 0.2419707 0.3989423 0.2419707
pnorm(c(-1, 0, 1))        # Pr(x =< a)
#> [1] 0.1586553 0.5000000 0.8413447
pnorm(1, lower.tail = F) # Pr(x > a)
#> [1] 0.1586553
qnorm(0.5)
#> [1] 0
rnorm(5)
#> [1]  0.4005958 -1.3736935  0.4515183 -0.2697028  0.8013775
rnorm(5, mean = 5, sd = 0.5)
#> [1] 4.664983 5.421341 5.790715 4.169431 5.484794
```

Статистичні розподіли та випадкові числа

Симуляція біноміального розподілу може бути дещо більш криптичною на перший погляд. Функція біноміального розподілу \(B(n, p)\) має параметр \(n\), що відповідає кількості спроб та \(p\), що відповідає вірогідності успіху спроби. У біноміальній функції R параметру \(n\) відповідає аргумент функції `size`:

```r
rbinom(n = 1, size = 1, prob = 0.5)    # окремий випадок біном розподілу - розподіл Бернулл
#> [1] 1

rbinom(n = 1, size = 10, prob = 0.5)   # один повтор, десять спроб
#> [1] 3
rbinom(n = 10, size = 1, prob = 0.5)   # десять повторів по одній спробі
#>  [1] 0 1 0 1 0 0 0 0 1 0
rbinom(n = 10, size = 10, prob = 0.5) # десять повторів, кожен по десять спроб
#>  [1] 4 5 8 2 5 6 5 2 6 6
```

Для того, щоб звернутися до документації конкретної функції розподілу, необхідно викликати назву цієї функції з одним із доступних префіксів, наприклад `?dt` для розподілу Стьюдента. Деякі інші корисні розподіли:

- `dunif` - рівномірний розподіл
- `dpois` - розподіл Пуасона
- `dmultinom` - мультиноміальний розподіл
- `dgeom` - геометричний розподіл

Статистичні розподіли та випадкові числа

Отримати випадковий елемент з певного сету можливо за допомогою `sample(x, size)`

```
sample(100, 5)
#> [1] 62 89 35 95 29
sample(c("a", "b", "c"), 5, replace = TRUE)
#> [1] "a" "a" "b" "a" "c"
sample(c("a", "b", "c"), 5, replace = TRUE, prob = c(.9, .05, .05))
#> [1] "a" "a" "a" "a" "a"
```

Генерація випадкових чисел надто важлива, щоб залишати її напризволяще.*

Якщо Ви колись щось читали про рандомні числа, Ви вірогідно вже знаєте, що усе представлене вище насправді є псевдорандомними числами згенерованими по певному алгоритму. Тому у випадку використання симуляції у своїх розрахунках Вам необхідно заздалегідь назначити `seed`, що забезпечить відтворюваність даних.

```
set.seed(42069)

sample(c(letters, LETTERS), 10, replace = T)
#>  [1] "D" "H" "P" "j" "R" "O" "Z" "W" "S" "J"
```

За деталями про те як працює RNG у R до `help("Random")`, якщо Вам необхідна якась особлива схема рандомізованої вибірки для експерименту зацініть бібліотеку sampling

*назва публікації Covevou, R.R. (1969), що формально стала цитатою

NA, NaN, Inf, та NULL

`NULL` є репрезентацією null-об'єкту, завжди має довжину `0` та не може мати жодних атрибутів. Ви можете зустріти `NULL` як дефолтне значення аргументу деяких функцій, наприклад у функції `seq()` дефолтні значення аргументів `length.out` та `along.with` є `NULL` — це означає, що дані аргументи є *опціональним*, але при їх виклику будуть виконані певні додаткові розрахунки

```
c()
#> NULL
```

`Inf` та `-Inf` репрезентують +/- нескінченність відповідно, `NaN` репрезентує значення Not a Number. І з тим і іншим зазвичай можна зіштовхнутися виконуючи математичні операції

```
0 / 0
#> [1] NaN
log(0)
#> [1] -Inf
1 + Inf
#> [1] Inf

c(-1, -Inf, Inf, NaN) < 0 # зверніть увагу, що порівняння з NaN дає NA
#> [1]  TRUE  TRUE FALSE    NA
```

NA, NaN, Inf, та NULL

`NA` означає Non Available або Not Applicable — репрезентація відсутності значення на певній позиції. `NA` є логічною константою довжиною один, хоча у різних типах векторів `NA` завжди відображається як `NA`, формально існує своє `NA` для кожного типу вектору окрім `raw`

```r
c("a", NA, NA) # зверніть увагу, що NA без лапок
#> [1] "a" NA  NA
c(NA, 5, 7)
#> [1] NA  5  7
c(TRUE, NA, FALSE)
#> [1]  TRUE    NA FALSE
```

Недетерміноване значення логічно повертає недетермінований результат

```r
c(1, -2, NA) > 0
#> [1]  TRUE FALSE    NA

all(c(NA, 2, 4) > 0)
#> [1] NA
any(c(NA, 2, 4) > 0) # але
#> [1] TRUE
```

NA, NaN, Inf, та NULL

Внаслідок цього низка математичних та статистичних функцій будуть повертати `NA` у випадку якщо `NA` присутні у ваших даних

```r
set.seed(12345)
x <- sample(c(runif(10), rep(NA, 10)), 10)

mean(x)
#> [1] NA
sum(x)
#> [1] NA
```

Виправити це можливо змінивши дефолтне значення аргументу `na.rm`, що властивий багатьом функціям, на `TRUE`

```r
mean(x, na.rm = TRUE)
#> [1] 0.5978509
sum(x, na.rm = TRUE)
#> [1] 3.587106
```

Перевірити чи є якесь значення `NA` можливо викликом `is.na()`, що повертає вектор логічних значень

```r
is.na(x)
#>  [1] FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE  TRUE
anyNA(x)
#> [1] TRUE
```

Векторизація та ресайклінг

Висока ефективність та інтуїтивність роботи з даними на R досягається можливістю векторизації низки базових математичних та логічних операцій, що дозволяє уникати написання циклів

Векторизація забезпечує виконання операції для кожного елементу вектору:

```r
x <- c(1, 2, 3)
x + 1 # до елементу x[[i]] додати 1
#> [1] 2 3 4
```

У випадку двох векторів довжина яких є рівною та більшою за 1 операція буде виконуватися попарно:

```r
x <- c(1, 2, 3)
y <- c(2, 4, 6)
x * y # елемент x[[i]] помножити на елемент y[[i]]
#> [1]  2  8 18
```

Якщо вектори мають різну довжину над коротшим вектором буде проведено ресайклінг, умовно кажучи, його буде подовжено до розміру довшого вектору для виконання операції:

```r
x <- c(1, 2, 3, 4)
y <- c(1, 0)
x * y # імпліцитно y буде перероблено як (1, 0, 1, 0)
#> [1] 1 0 3 0

y <- c(1, 0, 1)
x * y # імпліцитно y буде перероблено як (1, 0, 1, 1), попередження у консолі
#> [1] 1 0 3 4
```