

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 Scheme, 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 OS
- R поставляється зі своїм власним вбудованим CLI
- RStudio Desktop є вільним (ліцензія GNU AGPL) кросплатформним IDE початково розробленим специфічно під R (на сьогодні розширеном 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 зацікавитеся 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 функцій, 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 функціонування 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

```
1 # Бібліотека, що фактично складається лише з датасету пінгвінів Палмера
2 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


```
1 # DO NOT RUN
2 install.packages("BiocManager")
3 BiocManager::install("palmerpenguins") # лише для прикладу, Bioconductor не хостить даний пакет
```


Бібліотека також може бути встановлена безпосередньо із сурса


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

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

```
1 # install.packages("remotes")
2 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 tutorіали 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 [rdr](#) 

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 `?`

```
1  # ВИКЛИК ДОПОМОГИ ДЛЯ ФУНКЦІЇ ВИКЛИКУ ДОПОМОГИ
2
3  help("help") # " help is the primary interface to the help systems. "
4  ?help
5
6  help("?") # " Documentation Shortcuts - These functions provide access to documentation."
7  ?`?`      # Зверніть увагу на ` ` навколо оператора
8
9  help("??") # " Search the Help System - Allows for searching the help system for documentati
10 ?`??`      # 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

```
1 x <- 1
2 print(x) # виклик print() не є обов'язковим
3 #> [1] 1
4
5 y <- c(1, 2)
6 print(y)
7 #> [1] 1 2
8
9 z <- c(x, y)
10 print(z) # значення ідентичне print(c(x, y))
11 #> [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

```
1 a <- b <- c <- 6
2 print(a)
3 #> [1] 6
4
5 print(c(a, b, c))
6 #> [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

```
1 # install.packages("lobstr")
2 x <- c(1, 2, 4)
3 lobstr::obj_addr(x)
4 #> [1] "0x1c41e2ea938"
5
6 y <- x
7 lobstr::obj_addr(y)
8 #> [1] "0x1c41e2ea938"
```

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


```
1 x[[3]] <- 3
2 lobstr::obj_addr(x) # нова адреса
3 #> [1] "0x1c41e7d2348"
4
5 lobstr::obj_addr(y) # продовжує вказувати на стару адресу
6 #> [1] "0x1c41e2ea938"
7
8 print(x) # змінився
9 #> [1] 1 2 3
10 print(y) # все ще містить первісне значення
11 #> [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 behaviour** (що можна протестувати при взаємодії з нативним CLI R, результати отримані при інтерактивній взаємодії з консоллю 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:

```
1 erï <- "hi from Ukraine"
2 print(erï)
3 #> [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 :

```
1 ` _bad name ` <- "Just don't do things like this"
2 print(` _bad name `)
3 #> [1] "Just don't do things like this"
4 `if` <- "Don't do things like that either"
5 print(`if`)
6 #> [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")`

```
1 x + y
2 x - y
3 x * y
4 x / y
5 x ^ y  # можливо також використання синтаксису типу x ** y
6 x %% y  # modulo operation, повертає залишок від ділення
7 x %/% y # повертає цілу частину від ділення, округлення у меншу сторону
```

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

```
1 is_even <- function(x) {
2   if (x %% 2 != 0) {
3     cat("No,", x, "is an odd number")
4   } else {
5     cat("Yes,", x, "is an even number")
6   }
7 }
8
9 is_even(5)
10 #> No, 5 is an odd number
11
12 is_even(2)
13 #> Yes, 2 is an even number
```

Базові математичні функції

```
1 sqrt(x) # квадратний корінь
2 abs(x)  # абсолютне значення
3
4 cos(x) sin(x) tan(x) # та інші тригонометричні функції
5
6 sum(...) prod(...) # сума та продукт
7 min(...) max(...)  # мінімальне та максимальне значення
8
9 cumsum(x) cumprod(x) # кумулятивна сума та продукт
10 cummin(x) cummax(x)  # мініма та максима
11
12 log(x, base = exp(1)) log10() log2() # логарифми
13
14 round(x, digits = n) signif(x, digits = n) # округлення значень
15 trunc(x) floor(x) ceiling(x)
```

```
1 sum(1, 5, 10) # ... символізує, що ця функція може приймати довільну кількість аргументів
2 #> [1] 16
3 cumsum(c(1, 5, 10)) # ця функція приймає лише один аргумент, зверніть увагу на використання c()
4 #> [1] 1 6 16
5
6 # різниця між round() та signif()
7 round(0.005765, 3)
8 #> [1] 0.006
9 signif(0.005765, 3) # зверніть увагу що 5 округлюється у меншу сторону
10 #> [1] 0.00576
```

Бінарні логічні та реляційні оператори

Логічні оператори — `help("Logical")`

```
1  !x      # NOT, логічна негація
2
3  x & y    # AND, кон'юнкція
4  x && y
5
6  x | y    # OR, диз'юнкція
7  x || y
8
9  xor(x, y) # elementwise exclusive OR, виняткова диз'юнкція
10
11 isTRUE(x)
12 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

```
1  !(TRUE & FALSE) # правило де Моргана
2  #> [1] TRUE
3  !(TRUE | FALSE)
4  #> [1] FALSE
5
6  TRUE & (FALSE | TRUE) # дистрибутивність операцій
7  #> [1] TRUE
8  TRUE | (FALSE & TRUE)
9  #> [1] TRUE
```

Бінарні логічні та реляційні оператори

Реляційні оператори — `help("Comparison")`

```
1 x < y
2 x > y
3 x <= y
4 x >= y
5 x == y # еквівалентні
6 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

```
1 c(0, 2, 2, 1) > 0
2 #> [1] FALSE TRUE TRUE TRUE
3
4 any(c(0, 2, 2, 1) > 0)
5 #> [1] TRUE
6 all(c(0, 2, 2, 1) > 0)
7 #> [1] FALSE
```

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

```
1 1 == 1L # 1.00 дійсно є еквівалентним 1, але...
2 #> [1] TRUE
3 identical(1, 1L) # це два різних типи чисельних значень - double та integer
4 #> [1] FALSE
```

Оператори сетів

help("sets")

```
1 union(x, y)      # союз
2 intersect(x, y)   # перетин
3 setdiff(x, y)     # різниця (асиметрична)
4 setequal(x, y)    # чи є сеті рівнозначними
5
6 is.element(el, set) # чи належить елемент сету, ідентично команді x %in% y
```

```
1 x <- 1:6
2 y <- 4:10
3
4 union(x, y)
5 #> [1] 1 2 3 4 5 6 7 8 9 10
6 intersect(x, y)
7 #> [1] 4 5 6
8 setdiff(x, y)
9 #> [1] 1 2 3
10 setequal(x, y)
11 #> [1] FALSE
12 is.element(c(1, 4, 10), x)
13 #> [1] TRUE TRUE FALSE
14 "a" %in% y
15 #> [1] FALSE
16
17 # можливо менш очевидний приклад рівнозначності сетів
18 setequal(c(1, 1, 0), c(TRUE, TRUE, FALSE))
19 #> [1] TRUE
```

Bevare of float

```
1 x <- 2
2 y <- 2
3 x == y
4 #> [1] TRUE
5
6 y <- sqrt(y)^2
7 y
8 #> [1] 2
9
10 x == y # ???
11 #> [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

```
1 print(y, digits = 20)
2 #> [1] 2.00000000000000004441
```

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

```
1 all.equal(x, y) # забігаючи наперед, також можливо використовувати dplyr::near()
2 #> [1] TRUE
```

Якщо 53 знаки після коми не достатньо, то арифметичні операції довільної 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

```
1 mean(x) sd(x) # центральна тенденція - середнє та стандартне відхилення
2 mean(x) mad(x) # центральна тенденція - медіана та медіана абсолютних відхилень
3
4 quantile(x, probs) # квантиль, дефолтно повертає 0% 25% 50% 75% 100%
5 fivenum(x) # п'ятизначний підсумок Тьюкі
6 IQR(x) # інтерквантильний розмах
7 range(...) # мінімум та максимум
8
9 cor(x, y) cov(x, y) var(x) # кореляція, коваріація та варіація
```

Також доступними є низка статистичних тестів та моделей, зокрема:

```
1 t.test(x, ...) wilcox.test(x, ...) # Критерій Ст'юдента та Уїлкоксона/Менна-Уїтні
2 pairwise.t.test(x, g) pairwise.wilcox.test(x, g)
3
4 p.adjust(p, method) # корекція р-значень при багатократних порівняннях
5 TukeyHSD(x) # корекція р-значень методом Тьюкі при парних порівняннях
6
7 kruskal.test(x, g, ...) # Критерій Краскал-Уолліса
8 aov(formula, data) # дисперсійний аналіз, тип I
9
10 lm(formula, data) # лінійна модель
11 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

```
1 letters
2 #> [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
3 #> [20] "t" "u" "v" "w" "x" "y" "z"
4 LETTERS
5 #> [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
6 #> [20] "T" "U" "V" "W" "X" "Y" "Z"
```

назви місяців

```
1 month.abb
2 #> [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
3 month.name
4 #> [1] "January" "February" "March" "April" "May" "June"
5 #> [7] "July" "August" "September" "October" "November" "December"
```

та число π

```
1 pi
2 #> [1] 3.141593
```

Генерація регулярних сиквенсів

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

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

Послідовність від i до j з кроком u можна отримати командою `seq(...)`

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

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

```
1 rep(10, 3)
2 #> [1] 10 10 10
3 rep(c("a", "b", "c"), times = 3) # усю послідовність n разів
4 #> [1] "a" "b" "c" "a" "b" "c" "a" "b" "c"
5 rep(c("a", "b", "c"), each = 3) # кожен елемент послідовності n разів
6 #> [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 $N(0, 1)$

```
1 dnorm(c(-1, 0, 1))          # Pr(x = a)
2 #> [1] 0.2419707 0.3989423 0.2419707
3 pnorm(c(-1, 0, 1))          # Pr(x <= a)
4 #> [1] 0.1586553 0.5000000 0.8413447
5 pnorm(1, lower.tail = F)    # Pr(x > a)
6 #> [1] 0.1586553
7 qnorm(0.5)
8 #> [1] 0
9 rnorm(5)
10 #> [1] 0.22592121 -0.05349098 1.63891235 0.90918350 -0.77454147
11 rnorm(5, mean = 5, sd = 0.5)
12 #> [1] 4.307194 5.704443 4.933013 4.828275 4.950160
```

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**:

```
1 rbinom(n = 1, size = 1, prob = 0.5) # окремий випадок біном розподілу - розподіл Бернуллі
2 #> [1] 0
3
4 rbinom(n = 1, size = 10, prob = 0.5) # один повтор, десять спроб
5 #> [1] 1
6 rbinom(n = 10, size = 1, prob = 0.5) # десять повторів по одній спробі
7 #> [1] 0 0 1 0 0 1 0 1 1 0
8 rbinom(n = 10, size = 10, prob = 0.5) # десять повторів, кожен по десять спроб
9 #> [1] 4 3 6 3 8 5 5 4 6 7
```

Для того, щоб звернутися до документації конкретної функції розподілу, необхідно викликати назву цієї функції з одним із доступних префіксів, наприклад **?dt** для розподілу Стюдента. 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)`

```
1 sample(100, 5)
2 #> [1] 88 90 74 81 24
3 sample(c("a", "b", "c"), 5, replace = TRUE)
4 #> [1] "b" "a" "c" "a" "c"
5 sample(c("a", "b", "c"), 5, replace = TRUE, prob = c(.9, .05, .05))
6 #> [1] "a" "a" "a" "b" "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.

```
1 set.seed(42069)
2
3 sample(c(letters, LETTERS), 10, replace = T)
4 #> [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 Coveyou, 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

```
1 c()  
2 #> NULL
```

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

```
1 0 / 0  
2 #> [1] NaN  
3 log(0)  
4 #> [1] -Inf  
5 1 + Inf  
6 #> [1] Inf  
7  
8 c(-1, -Inf, Inf, NaN) < 0 # зверніть увагу, що порівняння з NaN дає NA  
9 #> [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 відображується як **NA**, формально існує своє **NA** для кожного типу вектору окрім **raw**

```
1 c("a", NA, NA) # зверніть увагу, що NA без лапок
2 #> [1] "a" NA NA
3 c(NA, 5, 7)
4 #> [1] NA 5 7
5 c(TRUE, NA, FALSE)
6 #> [1] TRUE NA FALSE
```

Недетерміноване значення логічно повертає недетермінований результат

```
1 c(1, -2, NA) > 0
2 #> [1] TRUE FALSE NA
3
4 all(c(NA, 2, 4) > 0)
5 #> [1] NA
6 any(c(NA, 2, 4) > 0) # але
7 #> [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

```
1 set.seed(12345)
2 x <- sample(c(runif(10), rep(NA, 10)), 10)
3
4 mean(x)
5 #> [1] NA
6 sum(x)
7 #> [1] NA
```

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

```
1 mean(x, na.rm = TRUE)
2 #> [1] 0.5978509
3 sum(x, na.rm = TRUE)
4 #> [1] 3.587106
```

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

```
1 is.na(x)
2 #> [1] FALSE TRUE FALSE FALSE TRUE FALSE FALSE FALSE TRUE TRUE
3 anyNA(x)
4 #> [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:

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

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

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

```
1 x <- c(1, 2, 3, 4)
2 y <- c(1, 0)
3 x * y # імпліцитно y буде перероблено як (1, 0, 1, 0)
4 #> [1] 1 0 3 0
5
6 y <- c(1, 0, 2)
7 x * y # імпліцитно y буде перероблено як (1, 0, 2, 1), попередження у консолі
8 #> [1] 1 0 6 4
```