# Podstawy programowania R

*Łukasz Wawrowski*

# Contents

**Wprowadzenie** **5**

**1 Wprowadzenie do R** **7**
1.1 R . . . 7
1.2 RStudio . . . 7
1.3 Ważne informacje . . . 7
1.4 Pakiety . . . 10
1.5 R jako kalkulator . . . 11

**2 Struktury danych** **13**
2.1 Wektor . . . 13
2.2 Macierz . . . 21
2.3 Czynnik . . . 25
2.4 Lista . . . 26
2.5 Ramka danych . . . 27
2.6 Rozwiązania do zadań . . . 29

**3 Przetwarzanie danych** **31**
3.1 Pakiet tidyverse . . . 31
3.2 Import danych . . . 32
3.3 Filtrowanie . . . 33
3.4 Wybieranie kolumn . . . 34
3.5 Tworzenie nowych zmiennych . . . 34
3.6 Zmiana nazwy zmiennej . . . 35
3.7 Podsumowanie danych . . . 35
3.8 Grupowanie . . . 35
3.9 Sortowanie . . . 37
3.10 Łączenie zbiorów . . . 38
3.11 Szeroka i wąska reprezentacja danych . . . 39
3.12 Eksport danych . . . 40
3.13 Zadania . . . 40
3.14 Case study . . . 40

**4 Wizualizacja danych** **41**
4.1 Wbudowane funkcje . . . 41
4.2 Pakiet ggplot2 . . . 41

**5 Programowanie w R** **45**
5.1 Funkcje . . . 45
5.2 Pętle . . . 45
5.3 Instrukcje warunkowe . . . 45

# Wprowadzenie

Literatura podstawowa:

- Przemysław Biecek - *Przewodnik po pakiecie R*
- Marek Gągolewski - *Programowanie w języku R. Analiza danych, obliczenia, symulacje.*
- Garret Grolemund, Hadley Wickham - *R for Data Science* (polska wersja)

Literatura dodatkowa:

- inne pozycje po polsku
- inne pozycje po angielsku

Internet:

- R-bloggers
- rweekly

# Chapter 1

# Wprowadzenie do R

GNU R to interpretowany język programowania oraz środowisko do obliczeń statystycznych i wizualizacji wyników [Wikipedia 2017].

Robert A. Muenchen - The Popularity of Data Science Software

## 1.1 R

Bazowa wersja R jest do pobrania ze strony r-project.org.

## 1.2 RStudio

RStudio to zintegrowane środowisko programistyczne (IDE) dla języka R dostępne za darmo na stronie RStudio.

Z R można także korzystać w Microsoft Visual Studio.

## 1.3 Ważne informacje

**R jest wrażliwy na wielkość liter.**

**Separatorem części dziesiętnej liczby jest kropka.**

**W codziennej pracy RStudio jest wygodniejsze, jednak długotrwałe obliczenia lepiej uruchamiać w trybie wsadowym w zwykłym R.**

- **Katalog roboczy**

Ważnym pojęciem w R jest katalog roboczy (ang. working directory), który określa gdzie zostaną zapisane pliki, wykresy, zbiory, itp. jeśli nie podamy dokładnej ścieżki do pliku. Katalog roboczy przypisuje się z wykorzystaniem funkcji `setwd("ścieżka do katalogu")`, a jego wartość można sprawdzić funkcją `getwd()`. W RStudio przypisanie katalogu roboczego odbywa się w momencie utworzenia projektu.

- **Projekt**

Katalog na dysku, w którym znajdują się wszystkie pliki projektu wraz z plikiem o rozszerzeniu .Rproj skojarzonym z RStudio.

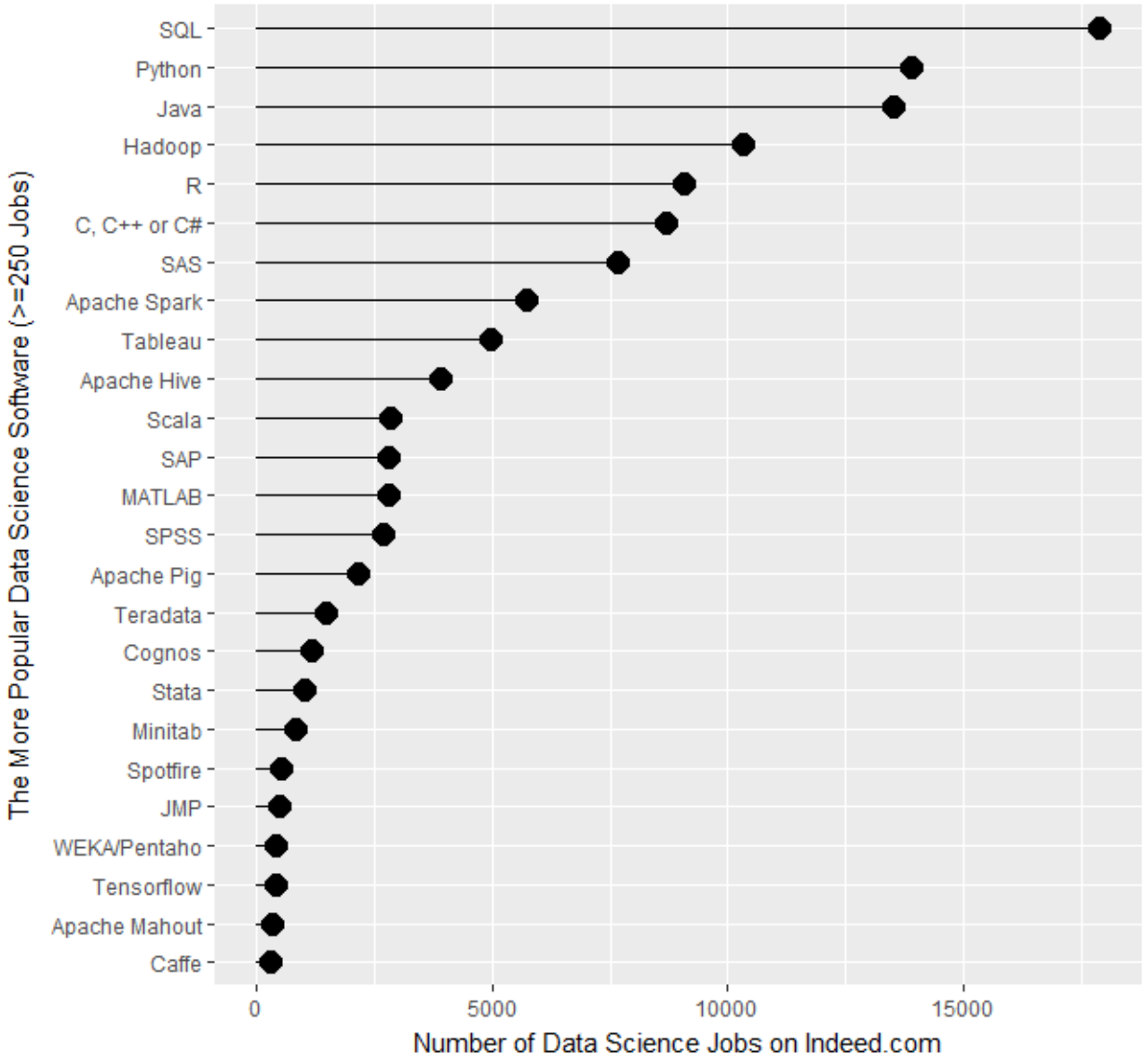

SQL
Python
Java
Hadoop
R
C, C++ or C#
SAS
Apache Spark
Tableau
Apache Hive
Scala
SAP
MATLAB
SPSS
Apache Pig
Teradata
Cognos
Stata
Minitab
Spotfire
JMP
WEKA/Pentaho
Tensorflow
Apache Mahout
Caffe
The More Popular Data Science Software (>=250 Jobs)
0
5000
10000
15000
Number of Data Science Jobs on Indeed.com

Figure 1.1:

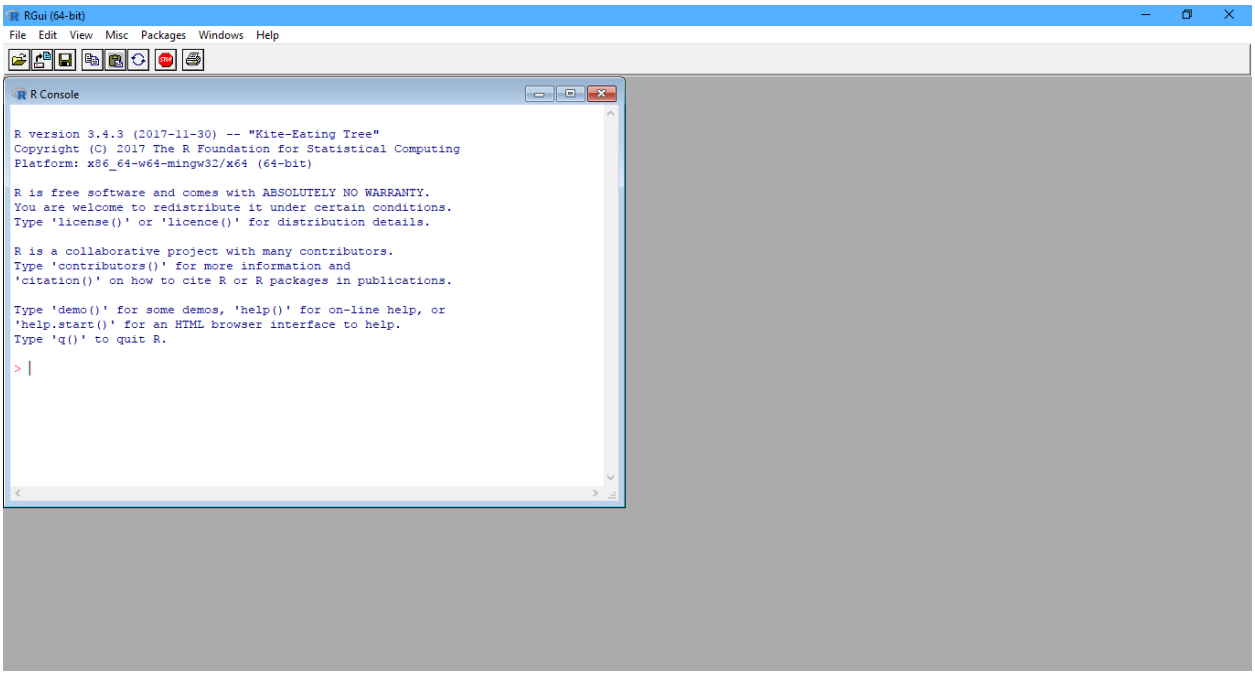



Figure 1.2:

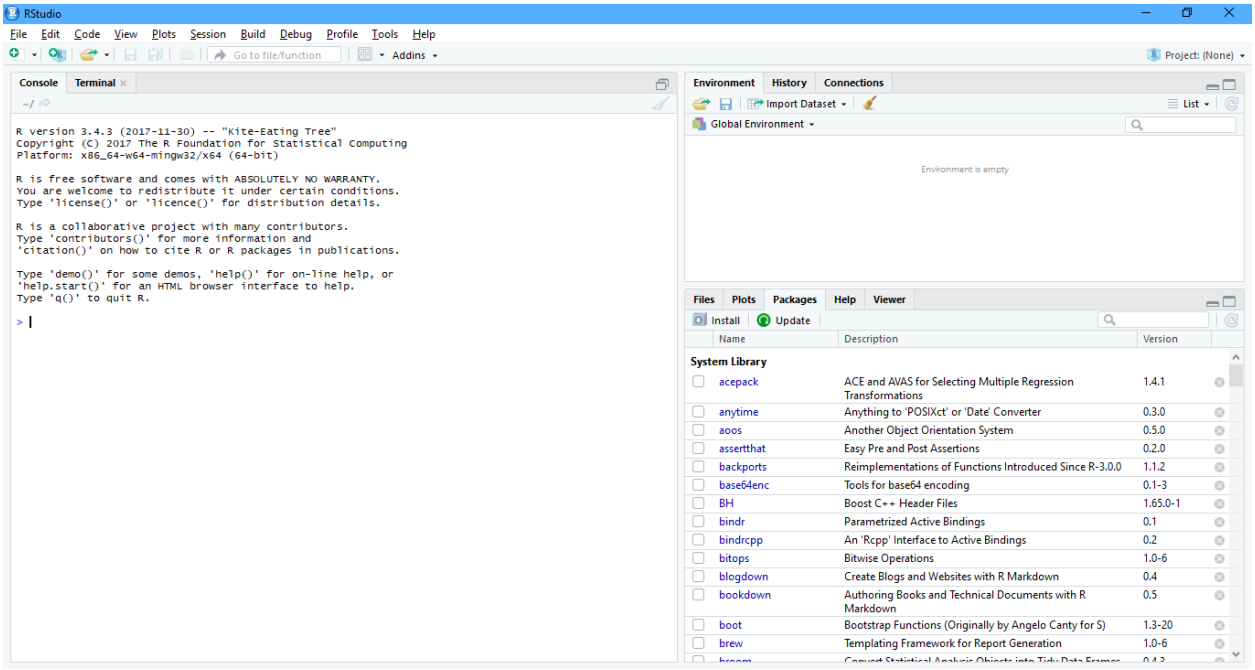



Figure 1.3:

- **Korzystanie z pomocy**

Dostęp do pomocy odnośnie wybranej funkcji można uzyskać na dwa sposoby. Pierwszym z nich jest poprzedzenie nazwy funkcji w konsoli znakiem zapytania np. `?getwd` lub wywołanie funkcji help na nazwie funkcji `help("getwd")`. Drugim sposobem jest umieszczenie kursora w dowolnym miejscu nazwy funkcji i wciśnięcie klawisza F1.

Internet - przede wszystkim stackoverflow.

- **Komentarze**

  Real programmers don't comment their code. If it was hard to write it should be hard to understand.

Dobrze napisany kod jest czytelny bez komentarzy. W R komentarze rozpoczynają się od symbolu #. Skrót klawiaturowy w RStudio to CTRL + SHIFT + C (do wstawiania i usuwania komentarzy).

- **Podpowiadanie składni**

RStudio ma zaimplementowaną funkcję podpowiadania składni. Listę możliwych funkcji i obiektów wywołuję się klawiszem TAB lub CTRL + SPACJA po wpisaniu co najmniej jednej litery. Kolejne naciśnięcie TAB lub ENTER powoduje uzupełnienie kodu o wybraną funkcję lub obiekt.

- **Wykonywanie programów**

Programy w R możemy tworzyć jako skrypty w pliku tekstowym o rozszerzeniu .R lub wywoływać polecenia bezpośrednio w konsoli. Kod programu napisanego w skrypcie przekazywany jest do konsoli. Gotowość do pracy R sygnalizuje w konsoli znakiem zachęty `>`. Jeśli podczas wykonywania programu w konsoli pojawi się znak `+` to oznacza oczekiwanie na kompletny kod - brak domkniętego nawiasu, cudzysłowia, itp.:

```
> getwd(
+
```

W powyższym przykładzie brakuje prawego nawiasu. Dodanie brakującego kodu spowoduje wykonanie przekazanego polecenia. Z kolei wciśnięcie klawisza ESC spowoduje przerwanie wykonywanie programu i powrót do znaku zachęty. Zawartość konsoli można wyczyścić stosując kombinację klawiszy CTRL + L.

- **Pliki**

Jeśli w pamięci znajdują się jakieś obiekty (zakładka Environment) to RStudio przy zamykaniu programu zapyta o zapisanie tych obiektów do pliku .RData. Jeżeli zdecydujemy się na tą propozycję to po ponownym uruchomieniu projektu obiekty znajdujące się w pliku .RData zostaną automatycznie wczytane do pamięci.

Można także samodzielnie tworzyć pliki o rozszerzeniu .RData z wykorzystaniem funkcji `save()`:

```
save(obiekt1, obiekt2, obiekt3, file = "nazwa_pliku.RData")
```

Wczytanie obiektów z takiego pliku do pamięci odbywa się z zastosowaniem funkcji `load()`:

```
load("nazwa_pliku.RData")
```

## 1.4 Pakiety

Podstawowe możliwości R są dosyć ograniczone. Rozszerzają je pakiety, których obecnie jest ponad 12 tysięcy. Można je przeglądać według kategorii w CRAN Task Views lub w wygodnej wyszukiwarce METACRAN i rdrr.io.

## 1.5 R jako kalkulator

Działania matematycznie w R:

| Operator | Operacja |
|---|---|
| + | dodawanie |
| - | odejmowanie |
| | mnożenie |
| / | dzielenie |
| ^ lub ** | potęgowanie |
| sqrt() | pierwiastkowanie |

W R istnieje także stała wbudowana `pi` przechowująca wartość liczby pi.

Funkcja `factorial(x)` zwraca silnię (znak wykrzyknika !) z podanej wartości x, a `sign(x)` sprawdza znak wyrażenia i zwraca odpowiednio wartość -1 jeśli wyrażenie jest ujemne, 0 jeśli jest równe 0 i 1 dla wyrażeń dodatnich.

Funkcja `exp(x)` zwraca wartość wyrażenia $e^x$, natomiast funkcja `log(x)` zwraca logarytm z podanej liczby. Domyślnie jest to logarytm naturalny, ale można zmienić podstawę podając wartość argumentu `base`.

Funkcja `abs(x)` zwraca wartość bezwzględną (absolutną) wyrażenia.

**Ćwiczenie**

Oblicz wartość wyrażenia: $2 \cdot \sqrt{\pi} + log_2 8$.

Rozwiązanie:

```
2*sqrt(pi)+log(8,2)
```

```
## [1] 6.544908
```

**Zadania**

Oblicz wartość wyrażeń:

1. $\frac{2^3 \cdot 6^2}{(\frac{1}{2})^2 \cdot (\frac{4}{5})^3}$
2. $\sqrt[3]{\frac{6-3.5}{2^{11}}}$
3. $\pi + \sqrt{e^4}$
4. $5! - log_{10} 100$
5. $|1 - e|$

# Chapter 2

# Struktury danych

W R praktycznie wszystko jest obiektem. Może to być zbiór danych, ale także wykres czy mapa. Zasadnicza różnica to klasa tych obiektów i operacje jakie mogą być na nich wykonywane.

Funkcje w R wymagają jako argumentów określonych typów obiektów - stąd tak ważna jak znajomość istniejących struktur.

Każdy obiekt w R możemy przypisać do tzw. obiektu nazwanego. Wówczas jest przechowywany w pamięci i można się do niego odwołać. Przypisanie odbywa się za pomocą operatora `<-`.

```
nazwa <- obiekt
obiekt -> nazwa
```

Można także przypisywać obiekty z wykorzystaniem znaku równości `=`, ale nie jest to zalecane ponieważ symbol ten jest używany w innych miejscach np. do deklarowania wartości argumentów w funkcji.

W R dostępna jest funkcja `assign`, która także umożliwia przypisanie nazwy do obiektu:

```
assign("nazwa", obiekt)
```

## 2.1 Wektor

Wektor jest najprostszym typem danych w R. Najczęściej korzysta się z trzech typów wektorów:

- logicznych
- liczbowych
- tekstowych

Wektor tworzy się z wykorzystaniem funkcji `c()`.

### 2.1.1 Wektor wartości logicznych

Przyjmuje wartości *prawda* lub *fałsz*:

```
c(TRUE, FALSE, FALSE)
```

```
## [1]  TRUE FALSE FALSE
```

lub w skróconej wersji:

```
c(T, F, F)
```

```
## [1]  TRUE FALSE FALSE
```

Do sprawdzenia długości wektora używa się funkcji `length`:

```
length(c(T, F, F))
```

```
## [1] 3
```

lub korzystając z obiektu nazwanego:

```
wart_log <- c(T,F,F)
length(wart_log)
```

```
## [1] 3
```

Wektory można także utworzyć poprzez replikację określonej wartości lub wektora z wykorzystaniem funkcji `rep`. Funkcja ta przyjmuje co najmniej dwa argumenty: obowiązkowo `x` - wektor wejściowy oraz jeden z następujących: `times` - liczba powtórzeń elementów wektora `x`, `each` - liczba powtórzeń elementów wektora `x` (wyjaśnienie różnicy poniżej) lub `length.out` - oczekiwana długość wektora wynikowego.

Trzy równoważne zapisy:

```
rep(x = c(T,F), times = 3)
```

```
## [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE
```

```
rep(c(T,F), times = 3)
```

```
## [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE
```

```
rep(c(T,F), 3)
```

```
## [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE
```

A tak to wygląda z argumentem `each`:

```
rep(c(T,F), each = 3)
```

```
## [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE
```

Wykorzystanie argumentu `length.out`:

```
rep(c(T,F), length.out = 5)
```

```
## [1]  TRUE FALSE  TRUE FALSE  TRUE
```

### 2.1.2 Wektor wartości liczbowych

W wektorze możemy przechowywać także liczby:

```
c(1, 3, -5, 2.5, .6) # nie trzeba pisać zera przed ułamkiem
```

```
## [1]  1.0  3.0 -5.0  2.5  0.6
```

Połączenie dwóch wektorów to także wektor:

```
c(c(1,2,3), c(3.5,4,4.5))
```

```
## [1] 1.0 2.0 3.0 3.5 4.0 4.5
```

Pojedyncza liczba też jest jednoelementowym wektorem:

```
length(2)
```

```
## [1] 1
```

Proste ciągi o różnicy równej 1 można generować wykorzystując dwukropek:

```
1:10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

lub

```
c(-5:-1,1:5)
```

```
##  [1] -5 -4 -3 -2 -1  1  2  3  4  5
```

Do generowania ciągów liczbowych o różnych różnicach wykorzystuje się funkcję `seq`, która przyjmuje następujące argumenty. Wartość początkową `from`, wartość końcową `to` oraz jeden z następujących: `by` - krok lub `length.out` - oczekiwana długość wektora.

To samo co `1:10`

```
seq(1, 10, 1)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Wartości niecałkowite:

```
seq(1, 2, 0.2)
```

```
## [1] 1.0 1.2 1.4 1.6 1.8 2.0
```

Wektor wartości malejących:

```
seq(10, 1, by=1) # błędny zapis
```

```
## Error in seq.default(10, 1, by = 1): wrong sign in 'by' argument
```

```
seq(10, 1, by=-1) # poprawny zapis
```

```
##  [1] 10  9  8  7  6  5  4  3  2  1
```

Tworzenie wektora w oparciu o argument `length.out` - funkcja sama dobiera krok:

```
seq(1, 7, length.out = 13)
```

```
##  [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0
```

Do generowania liczb pseudolosowych służy funkcja `runif(n)`, która do poprawnego wywołania wymaga tylko jednego argumentu - długości wektora wynikowego. Domyślnie losowane są liczby z przedziału $[0; 1]$ (tak jak w funkcji `los()` w Excelu), można to jednak zmienić podając odpowiednie wartości argumentów `min` i `max`.

```
runif(6)
```

```
## [1] 0.6666209 0.2487541 0.0660811 0.3471552 0.2560654 0.2310322
```

Obserwacje można także generować z innych rozkładów:

- `rnorm` - rozkład normalny,
- `rchisq` - rozkład $\chi^2$,
- `rt` - rozkład t-studenta,
- itp.

Wykaz wszystkich dostępnych w R rozkładów uzyskamy wywołując polecenie `help("Distributions")`.

Za każdym uruchomieniem jednej z wymienionych wyżej funkcji losujących wartości z danego rozkładu otrzymamy inne wartości:

```
runif(5)
```

```
## [1] 0.2922356 0.4974030 0.1838886 0.8267032 0.7709177
```

```
runif(5)
```

```
## [1] 0.4185505 0.7628634 0.2373476 0.4794877 0.5926578
```

Powtarzalność wyników możemy uzyskać ustalając ziarno generatora:

```
set.seed(123)
runif(5)
```

```
## [1] 0.2875775 0.7883051 0.4089769 0.8830174 0.9404673
```

```
set.seed(123)
runif(5)
```

```
## [1] 0.2875775 0.7883051 0.4089769 0.8830174 0.9404673
```

### 2.1.3 Wektor wartości tekstowych

W wektorze może być przechowywany tekst - wówczas poszczególne elementy zapisujemy w cudzysłowie lub apostrofach:

```
c("ala", "ma", "kota")
```

```
## [1] "ala"  "ma"   "kota"
```

```
c('ala', 'ma', 'kota')
```

```
## [1] "ala"  "ma"   "kota"
```

W RStudio wygodniej używać cudzysłowu, ponieważ program automatycznie go zamyka.

Istnieje także stała zawierająca litery alfabetu:

```
letters
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
## [18] "r" "s" "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"
## [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

### 2.1.4 Przeciążanie wektora

Jeśli w wektorze pomieszamy kilka typów zmiennych to R przekształci poszczególne wartości, tak aby stracić jak najmniej informacji:

```
c(TRUE, 2, 5)
```

```
## [1] 1 2 5
```

```
c(3, "cztery", 5)
```

```
## [1] "3"      "cztery" "5"
```

W pierwszym przypadku wartość `TRUE` została przekształcona na odpowiednik liczbowy - 1. Z kolei w drugim przykładzie podane liczby zostały przekonwertowane na tekst.

### 2.1.5 Operacje na wektorach

Na wektorach logicznych i liczbowych można wykonywać operacje arytmetyczne np. mnożenie:

```
1:10*2
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

Wektor liczbowy plus wektor liczbowy:

```
1:10 + c(1,2)
```

```
##  [1]  2  4  4  6  6  8  8 10 10 12
```

Wektor liczbowy razy wektor liczbowy:

```
1:10 * c(1,2)
```

```
##  [1]  1  4  3  8  5 12  7 16  9 20
```

Wektor liczbowy razy wektor logiczny:

```
1:10 * c(T, F)
```

```
##  [1] 1 0 3 0 5 0 7 0 9 0
```

Długości obu wektorów muszą być odpowiednie:

```
1:10 * c(T,F,T)
```

```
## Warning in 1:10 * c(T, F, T): longer object length is not a multiple of
## shorter object length
```

```
##  [1]  1  0  3  4  0  6  7  0  9 10
```

Dłuższy z wektorów musi być wielokrotnością krótszego.

Siłą rzeczy działania arytmetyczne na wektorach tekstowych nie są możliwe:

```
c("jeden", "dwa", "trzy", "cztery") * c(T,F)
```

```
## Error in c("jeden", "dwa", "trzy", "cztery") * c(T, F): non-numeric argument to binary operator
```

```
c("jeden", "dwa", "trzy", "cztery") + c(1,2)
```

```
## Error in c("jeden", "dwa", "trzy", "cztery") + c(1, 2): non-numeric argument to binary operator
```

### 2.1.6 Operacje agregujące

Na wektorach można także wykonywać operacje agregujące:

| Funkcja | Działanie |
|---|---|
| mean() | średnia elementów |
| sum() | suma elementów |
| prod() | iloczyn elementów |
| var() | wariancja elementów |
| sd() | odchylenie standardowe elementów |
| median() | mediana elementów |
| quantile() | kwantyl danego rzędu |
| min() | minimum |
| max() | maksimum |

Obliczenie skośności i kurtozy jest możliwe po zainstalowaniu pakietu `e1071`. Wówczas mamy dostęp do funkcji:

| Funkcja | Działanie |
|---|---|
| skewness() | skośność elementów |
| kurtosis() | kurtoza elementów |

Suma wektora numerycznego:

```
sum(1:10)
```

```
## [1] 55
```

Suma i średnia wektora logicznego:

```
sum(c(T, F, F, T))
```

```
## [1] 2
```

```
mean(c(T, F, F, T))
```

```
## [1] 0.5
```

Korzystanie z funkcji pochodzących z pakietów zewnętrznych wymaga wskazania skąd pochodzi dana funkcja. Można to zrobić na dwa sposoby: funkcją `library(pakiet)` - wówczas wszystkie funkcje z tego pakietu są wczytywane do pamięci i można do nich sięgać bezpośrednio lub wskazując przed nazwą funkcji z jakiego pakietu pochodzi.

Wczytanie pakietu:

```
library(e1071)
skewness(c(1,2,3,4,5,7,9,11,13))
```

```
## [1] 0.3451259
```

lub równoważnie:

```
e1071::skewness(c(1,2,3,4,5,7,9,11,13))
```

```
## [1] 0.3451259
```

Podsumowanie rozkładu wektora można także uzyskać z wykorzystaniem funkcji `summary(x)`:

```
summary(1:10)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.00    3.25    5.50    5.50    7.75   10.00
```

Działa także na wektorach tekstowych:

```
summary(c("jeden", "dwa", "trzy", "cztery"))
```

```
##    Length     Class      Mode
##         4 character character
```

### 2.1.7 Sprawdzanie typu wektora

Do określenia typu wektora służy funkcja `typeof`, `class` lub `mode`.

```
typeof(wart_log)
```

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

Sprawdzenie czy obiekt jest danego typu odbywa się z wykorzystaniem dedykowanych funkcji z przyrostkiem `is.`

```
is.logical(wart_log)
```

```
## [1] TRUE
```

```
is.character(wart_log)
```

```
## [1] FALSE
```

### 2.1.8 Rzutowanie wektorów

Czasami jako np. argument funkcji będzie wymagany inny typ wektora aniżeli aktualnie posiadany w pamięci. Można wówczas spróbować przekształcić taki wektor z wykorzystaniem funkcji rozpoczynającej się od `as.`:

```
typeof(wart_log)
```

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

```
as.numeric(wart_log)
```

```
## [1] 1 0 0
```

```
typeof(as.numeric(wart_log))
```

```
## [1] "double"
```

### 2.1.9 Indeksowanie wektorów

Aby uzyskać dostęp do części wektora korzysta się z indeksatora w postaci nawiasów kwadratowych. Utworzymy nowy wektor zawierający liczby całkowite od 10 do 20:

```
wart_10_20 <- seq(10,20)
wart_10_20
```

```
##  [1] 10 11 12 13 14 15 16 17 18 19 20
```

a następnie wybieramy trzecią obserwację:

```
wart_10_20[3]
```

```
## [1] 12
```

Możemy także odwołać się do większego zakresu:

```
wart_10_20[3:5]
```

```
## [1] 12 13 14
```

I wybranych elementów:

```
wart_10_20[c(1,3,5)]
```

```
## [1] 10 12 14
```

W ten sposób można także modyfikować odpowiednie elementy wektora:

```
wart_10_20[7] <- 90
```

Wybór obserwacji większych od 15:

```
wart_10_20[wart_10_20>15]
```

```
## [1] 90 17 18 19 20
```

Z kolei następujący zapis zwróci nam wektor wartości logicznych:

```
wart_10_20 > 15
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

### 2.1.10 Wartości nieliczbowe

Brak danych w R jest przedstawiany jako wartość `NA` (ang. *not available*) i może powodować trudności z wywoływaniem niektórych funkcji:

```
v_na <- c(1,2,1,NA,1)
v_na
```

```
## [1]  1  2  1 NA  1
```

```
sum(v_na)
```

```
## [1] NA
```

W związku z tym większość funkcji ma zaimplementowany dodatkowy argument służący do obsługi tego typu wartości, który najczęściej nie uwzględnia tych wartości w obliczeniach:

```
sum(v_na, na.rm = TRUE)
```

```
## [1] 5
```

Oprócz braku danych podczas obliczeń możemy natrafić na wartości nieokreślone `NaN` (ang. *not a number*) oraz nieskończone `Inf` (ang. *infinity*).

```
0/0
```

```
## [1] NaN
```

```
1/0
```

```
## [1] Inf
```

```
sqrt(-10)
```

```
## Warning in sqrt(-10): NaNs produced
```

```
## [1] NaN
```

W R istnieje także wartość `NULL`, która jest podstawowym typem danych a nie wartością. `NULL` można traktować jako odpowiednik zbioru pustego. Jest stosowany np. w funkcjach, które niczego nie zwracają.

```
v_null <- c(1,2,1,NULL,1)
v_null
```

```
## [1] 1 2 1 1
```

```
sum(v_null)
```

```
## [1] 5
```

### 2.1.11 Zadania

1. Ile wynosi suma elementów większych od 10 dla następujących liczb: `12, 5, 20, 18, 8.5, 10, 4, 101, -2`?
2. Z wykorzystaniem funkcji `seq` i na podstawie wektora … dokonaj przekształcenia tworząc następujący wektor: `2 0 0 4 0 0 6 0 0 8 0 0`.
3. Dane są dwa wektory - a: `2, 3, 7, 8, 2`, b: `9, 1, 2, 0, 2`. Jakiego typu będzie wektor będący wynikiem działania `a<=b`?
4. Uzupełnij wektor `letters` o polskie litery diakrytyzowane. Jaką długość ma nowo utworzony wektor?
5. Wylosuj z rozkładu normalnego 1000 obserwacji z ziarnem równym 76. Ile wynosi kurtoza tych wartości?

## 2.2 Macierz

Macierze są wykorzystywane w R do przechowywania np. odległości pomiędzy punktami czy wskazywania sąsiedztwa obszarów geograficznych.

Do tworzenia macierzy służy funkcja `matrix`:

```
m <- matrix(1:6, nrow = 2, ncol=3)
m
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

Z wykorzystaniem wybranych funkcji można sprawdzić wymiary macierzy, liczbę wierszy oraz kolumn:

```
dim(m)
```

```
## [1] 2 3
```

```
ncol(m)
```

```
## [1] 3
```

```
nrow(m)
```

```
## [1] 2
```

Macierz może także zawierać tekst:

```
matrix(letters[1:9], nrow=3)
```

```
##      [,1] [,2] [,3]
## [1,] "a"  "d"  "g"
## [2,] "b"  "e"  "h"
## [3,] "c"  "f"  "i"
```

Domyślnie macierz układana jest kolumnami. Aby to zmienić należy dodać argument `byrow=TRUE`:

```
matrix(letters[1:9], nrow=3, byrow=TRUE)
```

```
##      [,1] [,2] [,3]
## [1,] "a"  "b"  "c"
## [2,] "d"  "e"  "f"
## [3,] "g"  "h"  "i"
```

Jeśli liczba elementów wejściowych jest mniejsza iloczyn podanej liczby kolumn i wierszy to w brakujące miejsce wstawiane są elementy z początku wektora wejściowego:

```
matrix(letters[1:7], nrow=3, byrow=TRUE)
```

```
## Warning in matrix(letters[1:7], nrow = 3, byrow = TRUE): data length [7] is
## not a sub-multiple or multiple of the number of rows [3]
```

```
##      [,1] [,2] [,3]
## [1,] "a"  "b"  "c"
## [2,] "d"  "e"  "f"
## [3,] "g"  "a"  "b"
```

Z kolei macierz diagnonalną posiadającą elementy niezerowe wyłącznie na przekątnej tworzy się z wykorzystaniem funkcji `diag`. Macierz jednostkowa o wymiarach $4 \times 4$:

```
diag(4)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    0    1    0    0
## [3,]    0    0    1    0
## [4,]    0    0    0    1
```

Macierz diagonalna o wartościach 5 na przekątnej i wymiarach $3 \times 3$

```
diag(5, nrow=3, ncol=3)
```

```
##      [,1] [,2] [,3]
## [1,]    5    0    0
## [2,]    0    5    0
## [3,]    0    0    5
```

Funkcja `diag` umożliwia także ekstrakcję przekątnej z istniejącej już macierzy:

```
diag(matrix(letters[1:9], nrow=3))
```

```
## [1] "a" "e" "i"
```

### 2.2.1 Łączenie macierzy

Z wykorzystaniem funkcji `rbind` i `cbind` można odpowiednio łączyć obiekty wierszami (ang. *row bind*) lub kolumnami (ang. *col bind*):

```
rbind(m, c(99, 88, 77))
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
## [3,]   99   88   77
```

```
cbind(m, matrix(101:104, nrow=2))
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5  101  103
## [2,]    2    4    6  102  104
```

### 2.2.2 Indeksowanie macierzy

Dostęp do poszczególnych elementów macierzy odbywa się z wykorzystaniem nawiasów kwadratowych, ale można podać dwie wartość - `obiekt[wiersz,kolumna]`:

```
m[2,1] # drugi wiersz, pierwsza kolumna
```

```
## [1] 2
```

```
m[2,]  # tylko drugi wiersz
```

```
## [1] 2 4 6
```

```
m[,1]  # tylko pierwsza kolumna
```

```
## [1] 1 2
```

```
m[,]   # wszystkie obserwacje
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
m[]    # wszystkie obserwacje
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

W ten sposób można dokonać modyfikacji konkretnych elementów macierzy:

```
m[2,1] <- 77
m
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]   77    4    6
```

### 2.2.3 Operacje na macierzach

Na macierzach można wywołać szereg operacji:

| Operator/funkcja | Działanie |
|---|---|
| a %*% b | mnożenie macierzy a i b |
| t(a) | transpozycja macierzy a |
| det(a) | wyznacznik macierzy a |
| solve(a) | macierz odwrotna z a |
| solve(a, b) | rozwiązanie układu a*x=b |

Rozważmy dwie macierze:

```
a <- matrix(c(2, 3, 4, 2, 1, 2, 1, 3, 2), nrow = 3)
b <- matrix(6:1, ncol=2)
a;b
```

```
##      [,1] [,2] [,3]
## [1,]    2    2    1
## [2,]    3    1    3
```

```
## [3,]    4    2    2
```

```
##      [,1] [,2]
## [1,]    6    3
## [2,]    5    2
## [3,]    4    1
```

Aby przeprowadzić mnożenie macierzy `a` i `b`, liczba kolumn macierzy `a` musi być równa liczbie wierszy w macierzy `b`. Z kolei rozmiar macierzy wyjściowej to liczba wierszy macierzy `a` i liczba kolumn macierzy `b`.

```
a %*% b
```

```
##      [,1] [,2]
## [1,]   26   11
## [2,]   35   14
## [3,]   42   18
```

Transpozycja macierzy `b`:

```
t(b)
```

```
##      [,1] [,2] [,3]
## [1,]    6    5    4
## [2,]    3    2    1
```

Wyznacznik macierzy `a`:

```
det(a)
```

```
## [1] 6
```

Macierz odwrotna do macierzy `a`:

```
solve(a)
```

```
##            [,1]       [,2]       [,3]
## [1,] -0.6666667 -0.3333333  0.8333333
## [2,]  1.0000000  0.0000000 -0.5000000
## [3,]  0.3333333  0.6666667 -0.6666667
```

Wyznaczenie macierzy `x` w równaniu `a*x=b`:

```
solve(a,b)
```

```
##           [,1]      [,2]
## [1,] -2.333333 -1.833333
## [2,]  4.000000  2.500000
## [3,]  2.666667  1.666667
```

```
a %*% solve(a,b)
```

```
##      [,1] [,2]
## [1,]    6    3
## [2,]    5    2
## [3,]    4    1
```

```
b
```

```
##      [,1] [,2]
## [1,]    6    3
## [2,]    5    2
## [3,]    4    1
```

### 2.2.4 Zadanie

1. Co powstanie po przemnożeniu macierzy przez jej macierz odwrotną?
2. Estymator parametrów beta w metodzie najmniejszych kwadratów jest dany wzorem:

$$b = (X'X)^{-1}X'y$$

Zmienna $x_1$ przyjmuje wartości `2,4,1,6,9,3,2,9,10,7`, zmienna $x_2$ `1.5,0.2,0.1,2,3.1,1.2,0.4,2.9,2.5,1.9`, a zmienna $x_0$ to wektor jedynek. Te trzy zmienne tworzą macierz $X$. Z kolei wartości zmiennej $y$ są następujące `12,15,10,19,26,13,13,21,29,18`. Wyznacz wartość $b$.

```
zad1 <- matrix(1:4, nrow=2)
solve(zad1)
```

```
##      [,1] [,2]
## [1,]   -2  1.5
## [2,]    1 -0.5
```

```
zad1 %*% solve(zad1)
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

## 2.3 Czynnik

Czynnik (ang. *factor*) służy do przechowywania danych jakościowych o mało licznej liczbie kategorii, mierzonych na skali nominalnej i porządkowej.

Rozważmy informacje o wykształceniu:

```
wyk <- rep(c("podstawowe", "średnie", "wyższe"), c(5,3,2))
wyk
```

```
##  [1] "podstawowe" "podstawowe" "podstawowe" "podstawowe" "podstawowe"
##  [6] "średnie"    "średnie"    "średnie"    "wyższe"     "wyższe"
```

i dokonajmy transformacji na czynnik:

```
wyk_f <- factor(wyk)
wyk_f
```

```
##  [1] podstawowe podstawowe podstawowe podstawowe podstawowe średnie
##  [7] średnie    średnie    wyższe     wyższe
## Levels: podstawowe średnie wyższe
```

Funkcja `summary()` wywołana na czynniku zwraca wynik innego typu aniżeli na wektorze tekstowym:

```
summary(wyk)
```

```
##    Length     Class      Mode
##        10 character character
```

```
summary(wyk_f)
```

```
## podstawowe    średnie     wyższe
##          5          3          2
```

Jeśli chcemy zaakcentować fakt, że zmienne są mierzone na skali porządkowej dodajemy argument `ordered=TRUE`:

```
wyk_of <- factor(wyk, ordered = TRUE)
wyk_of
```

```
##  [1] podstawowe podstawowe podstawowe podstawowe podstawowe średnie
##  [7] średnie    średnie    wyższe     wyższe
## Levels: podstawowe < średnie < wyższe
```

W łatwy sposób możemy edytować etykiety:

```
levels(wyk_of) <- c("pod.", "śr.", "wyż.")
wyk_of
```

```
##  [1] pod. pod. pod. pod. pod. śr.  śr.  śr.  wyż. wyż.
## Levels: pod. < śr. < wyż.
```

Czynniki mają szczególne znaczenie w przypadku tworzenia wykresów, gdy chcemy określić porządek wyświetlania.

## 2.4 Lista

Listy to ciągi złożone z elementów o dowolnych typach. Mogą przydać się w szczególności przy budowaniu funkcji, które zwracają tylko jedną wartość. Wówczas dane różnego typu mogą być zawarte w takiej liście.

Tworzenie prostej listy:

```
l <- list(TRUE, c(1,2,3,4), "element tekstowy")
l
```

```
## [[1]]
## [1] TRUE
##
## [[2]]
## [1] 1 2 3 4
##
## [[3]]
## [1] "element tekstowy"
```

Już na pierwszy rzut oka widać bardziej złożoną strukturę listy. W związku z tym odwoływanie do poszczególnych elementów będzie trochę się różnić od wektorów czy macierzy.

```
l[2] # druga lista
```

```
## [[1]]
## [1] 1 2 3 4
```

```
l[[2]] # zawartość listy
```

```
## [1] 1 2 3 4
```

```
l[[2]][3] # trzeci element wektora drugiej listy
```

```
## [1] 3
```

Listę można także rozwinąć do wektora z wykorzystaniem funkcji `unlist`:

```
unlist(l)
```

```
## [1] "TRUE"              "1"                 "2"
## [4] "3"                 "4"                 "element tekstowy"
```

Poszczególne elementy listy można nazwać:

```
ln <- list(log=TRUE, num=c(1,2,3,4), tekst="element tekstowy")
ln
```

```
## $log
## [1] TRUE
##
## $num
## [1] 1 2 3 4
##
## $tekst
## [1] "element tekstowy"
```

Wówczas można uzyskać do nich dostęp poprzez symbol `$` i podaną nazwę:

```
ln$num
```

```
## [1] 1 2 3 4
```

```
ln[[2]] # normalne indeksowanie nadal działa
```

```
## [1] 1 2 3 4
```

```
ln$num[2]
```

```
## [1] 2
```

## 2.5 Ramka danych

Ramka danych to tabela, która przypomina tą z Excela zawierająca dane o różnych typach. Tworzona za pomocą funkcji `data.frame`:

```
df <- data.frame(plec=c("m", "k", "k", "m", "k", "m", "m", "m"),
                 wzrost=c(173, 170, 163, 178, 169, 180, 175, NA),
                 pali=c(T, F, F, F, T, F, NA, T))
```

W RStudio po wybraniu tego obiektu w zakładce `Environment` pojawia się przyjazne okno do przeglądania oraz poglądowego filtrowania i sortowania danych ze zbioru.

Możemy zobaczyć podsumowanie całego zbioru wywołując na nim funkcję `summary()`:

```
summary(df)
```

```
##  plec      wzrost         pali
##  k:3   Min.   :163.0   Mode :logical
##  m:5   1st Qu.:169.5   FALSE:4
##        Median :173.0   TRUE :3
##        Mean   :172.6   NA's :1
##        3rd Qu.:176.5
##        Max.   :180.0
##        NA's   :1
```

Ramki danych można indeksować w taki sam sposób jak macierze lub z wykorzystaniem operatora `$`:

```
df[,2] # druga kolumna
```

```
## [1] 173 170 163 178 169 180 175  NA
```

```
df$wzrost # kolumna wzrost
```

```
## [1] 173 170 163 178 169 180 175  NA
```

```
df[,c("plec", "pali")]
```

```
##   plec  pali
## 1    m  TRUE
## 2    k FALSE
## 3    k FALSE
## 4    m FALSE
## 5    k  TRUE
## 6    m FALSE
## 7    m    NA
## 8    m  TRUE
```

Z kolei do wyboru obserwacji można wykorzystać warunek:

```
df[df$plec=="m",]
```

```
##   plec wzrost  pali
## 1    m    173  TRUE
## 4    m    178 FALSE
## 6    m    180 FALSE
## 7    m    175    NA
## 8    m     NA  TRUE
```

Wyodrębnienie informacji o wzroście tylko dla kobiet i wyznaczenie średniej:

```
wzrost_k <- df$wzrost[df$plec == "k"]
wzrost_k
```

```
## [1] 170 163 169
```

```
mean(wzrost_k)
```

```
## [1] 167.3333
```

Widzimy, że dla mężczyzn nie udało się ustalić wszystkich informacji i jeden z nich nie ma podanego wzrostu, a dla drugiego brakuje informacji o paleniu papierosów. Możemy usunąć braki danych w kolumnach korzystając z funkcji `complete.cases()`:

```
df[complete.cases(df$wzrost),] # tylko zmienna wzrost
```

```
##   plec wzrost  pali
## 1    m    173  TRUE
## 2    k    170 FALSE
## 3    k    163 FALSE
## 4    m    178 FALSE
## 5    k    169  TRUE
## 6    m    180 FALSE
## 7    m    175    NA
```

```
df[complete.cases(df),] # wszystkie zmienne
```

```
##   plec wzrost  pali
## 1    m    173  TRUE
## 2    k    170 FALSE
## 3    k    163 FALSE
```

```
## 4    m    178 FALSE
## 5    k    169  TRUE
## 6    m    180 FALSE
```

Zbiory danych przechowywane są także w R i pochodzą z różnych pakietów. Wywołując funkcję `data("zbior")` ładujemy dany zbiór do pamięci. Do szybkiego podglądu zebranych danych służy funkcja `head()`, która domyślnie wyświetla 6 pierwszych obserwacji ze zbioru:

```
data("iris")
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

### 2.5.1 Zadania

Załaduj do pamięci zbiór o nazwie ChickWeight.

1. Ile razy jedzenie otrzymał kurczak o numerze 15?
2. Ile wynosi mediana wagi kurczaka o numerze 35?
3. Ile średnio ważyły kurczaki na diecie nr 1, a ile na diecie nr 2?

## 2.6 Rozwiązania do zadań

### 2.6.1 Macierz

```
x1 <- c(2,4,1,6,9,3,2,9,10,7)
x2 <- c(1.5,0.2,0.1,2,3.1,1.2,0.4,2.9,2.5,1.9)
x0 <- rep(1,length(x1))
y <- c(12,15,10,19,26,13,13,21,29,18)

x <- cbind(x0, x1, x2)

b <- solve(t(x)%*%x)%*%t(x)%*%y
```

# Chapter 3

# Przetwarzanie danych

## 3.1 Pakiet tidyverse

Pakiet `tidyverse` to zestaw pakietów do kompleksowego przetwarzania i wizualizacji danych. Ładuje następujące pakiety:

- ggplot2 - tworzenie wykresów,
- dplyr - przetwarzanie danych,
- tidyr - zmiana reprezentacji danych,
- readr - wczytywanie danych tekstowych,
- purrr - programowanie funkcyjne
- tibble - sposób przechowywania danych,
- stringr - przetwarzanie tekstów,
- forcats - przetwarzanie faktorów

Manifest tidyverse ustala następujące zasady:

- powtórne użycie istniejących struktur danych,
- tworzenie czytelnych kodów z operatorem pipe `%>%` (ang. rura, przewód, łącznik).

Wobec tego załadujmy pakiet `tidyverse`:

```
library(tidyverse)
```

W konsoli pojawi się informacja o wersji załadowanych pakietów oraz o konfliktach występujących pomiędzy pakietami. Konflikty te wynikają z takich samych nazw funkcji w różnych pakietach. Kolejność wczytywania pakietów ma znaczenie - kolejny pakiet przykryje funkcje z wcześniej wczytanego. Wywołanie przykrytej funkcji jest możliwe poprzez zapis `nazwa_pakietu::nazwa_funkcji`.

Korzystanie z pakietu i zasad `tidyverse` to dużo bardziej czytelny kod w porównaniu do wbudowanych funkcji. Poniżej przedstawiony jest przykład przetwarzania danych polegający na filtrowaniu, wyborze kolumn oraz utworzeniu nowej zmiennej.

```
data("ChickWeight")

# bez pakietu tidyverse

chick_15 <- ChickWeight[ChickWeight$Chick=="15",]
chick_15 <- chick_15[c("weight", "Time", "Diet"),]
chick_15$weight_kg <- chick_15$weight/1000
```

```
# z pakietem tidyverse

chick_15 <- ChickWeight %>%
  filter(Chick=="15") %>%
  select(-Chick) %>%
  mutate(weight_kg=weight/1000)
```

Rozwiązanie z wykorzystaniem wbudowanych funkcji to 133 znaki, natomiast wykorzystanie `tidyverse` to 30% oszczędność miejsca i tylko 92 znaki.

## 3.2 Import danych

Wczytywanie danych do R jest możliwe z wielu różnych źródeł. Funkcje, które to umożliwiają zwykle mają nazwę rozpoczynającą się od `read`.

Będziemy korzystać z następujących zbiorów danych:

- movies - plik tekstowy zawierający informacje o filmach,
- bank - plik excel zawierający dane dot. kampanii marketingowej banku, opis zmiennych,
- rossmann - plik excel zawierający dane ze sklepów Rossmann,
- lotto - plik tekstowy zawierający dane z losowań Lotto.

### 3.2.1 Pliki CSV

Do wczytywania plików csv można wykorzystać wbudowaną funkcję `read.csv()` lub tą pochodzącą z pakietu `readr` - `read_csv()`. W obu przypadkach wynik wczytania będzie podobny.

```
movies <- read.csv("data/movies.csv")

movies2 <- read_csv("data/movies.csv")
```

```
## Parsed with column specification:
## cols(
##   title = col_character(),
##   genre = col_character(),
##   director = col_character(),
##   year = col_integer(),
##   duration = col_integer(),
##   gross = col_integer(),
##   budget = col_integer(),
##   cast_facebook_likes = col_integer(),
##   votes = col_integer(),
##   reviews = col_integer(),
##   rating = col_double()
## )
```

Jeśli nas plik ma nietypową strukturę to w funkcji `read.csv()` możemy określić dodatkowe argumenty informując o nazwach kolumn obecnych w pliku (`header =`), separatorze kolumn (`sep =`) lub separatorze miejsc dziesiętnych (`dec =`)

```
movies <- read.csv(file = "data/movies.csv", header = T, sep=",", dec=".")
```

### 3.2.2 Pliki excel

Do wczytywania plików z Excela niezbędny jest dodatkowy pakiet `readxl`. W funkcji `read_xlsx()` podajemy jako argument nazwę pliku. Możemy także dodać nazwę lub numer arkusza w argumencie (`sheet =`) oraz zakres komórek jako wartość argumentu `range =`.

```
library(readxl)

bank <- read_xlsx("data/bank.xlsx")

# bank <- read_xlsx("data/bank.xlsx", sheet = "dane")
# bank <- read_xlsx("data/bank.xlsx", sheet = 1)

bank_a1i30 <- read_xlsx("data/bank.xlsx", range = "A1:I30")

rossmann <- read_xlsx("data/rossmann.xlsx")
```

### 3.2.3 Pliki tekstowe

Z kolei do wczytywania plików tekstowych wykorzystuje się funkcję `read.table()`. Wczytywany plik nie musi być zlokalizowany na dysku twardym - może to być link internetowy.

```
lotto <- read.table("http://www.mbnet.com.pl/dl.txt")
names(lotto) <- c("lp", "data", "numery")
```

## 3.3 Filtrowanie

Do przetwarzania danych służą funkcje z pakietu `dplyr`. Większość z nich jako pierwszy argument przyjmuje przetwarzany zbiór danych, ale można tego uniknąć wykorzystując symbole `%>%`.

Filtrowanie polega na wybraniu obserwacji, które spełniają określony warunek lub warunki. Ze zbioru `movies` wybierzmy wszystkie komedie:

```
komedie <- filter(movies, genre=="Comedy")
```

lub alternatywnie:

```
komedie <- movies %>%
  filter(genre=="Comedy")
```

Po zmiennej, która jest filtrowana musimy podać operator porównania czyli podwójny znak równości `==`. Jeśli chcemy filtrować po większej liczbie zmiennych to kolejne warunki dodajemy po przecinku:

```
komedie_2012 <- movies %>%
  filter(genre=="Comedy", year==2012)
```

Wówczas oba warunki muszą zostać spełnione czyli pomiędzy nimi zachodzi relacja `i`. Równoważny zapis jest następujący:

```
komedie_2012 <- movies %>%
  filter(genre=="Comedy" & year==2012)
```

Pomiędzy warunkami może także zachodzić relacja `lub`. Wybieramy filmy, które są komediami **lub** miały swoją premierę w 2012 roku.

```
komedie_l_2012 <- movies %>%
  filter(genre=="Comedy" | year==2012)
```

Możliwy jest także wybór wielu kryteriów filtrowania poprzez operator `%in%`:

```
komedie_familijne <- movies %>%
  filter(genre %in% c("Comedy", "Family"))

movies_2000_2010 <- movies %>%
  filter(year %in% 2000:2010)
```

## 3.4 Wybieranie kolumn

Do wyboru kolumn służy funkcja `select()`. Zmodyfikujemy wcześniej utworzony zbiór `komedie`:

```
komedie <- movies %>%
  filter(genre=="Comedy") %>%
  select(title, year, duration, budget, rating)
```

Ten sam kod możemy zapisać zagnieżdżając funkcje, ale traci on w ten sposób na czytelności:

```
komedie <- select(filter(movies, genre=="Comedy"), title, year, duration, budget, rating)
```

Możemy także wskazać, które zmienne nie mają znaleźć się w zbiorze wynikowym:

```
komedie <- movies %>%
  filter(genre=="Comedy") %>%
  select(-genre)
```

Natomiast jeśli zmiennych jest więcej to musimy jest umieścić w wektorze, żeby nie pisać przed każdą zmienną znaku minus:

```
komedie <- movies %>%
  filter(genre=="Comedy") %>%
  select(-genre, -director, -gross, -budget)


komedie <- movies %>%
  filter(genre=="Comedy") %>%
  select(-c(genre, director, gross, budget))
```

Z wykorzystaniem znaku dwukropka możemy także wskazywać zakresy zmiennych:

```
komedie <- movies %>%
  filter(genre=="Comedy") %>%
  select(-genre, -c(gross:reviews))
```

## 3.5 Tworzenie nowych zmiennych

Do utworzenia nowej zmiennej wykorzystuje się funkcję `mutate()`. Utwórzmy w naszym zbiorze nową zmienną, która będzie zawierała czas trwania filmu w godzinach:

```
komedie <- movies %>%
  filter(genre=="Comedy") %>%
```

```
  select(-genre, -c(gross:reviews)) %>%
  mutate(dur_hour = duration/60)
```

Rozsądnie będzie zaokrąglić otrzymaną wartość do jednego miejsca po przecinku - służy do tego funkcja `round()`:

```
komedie <- movies %>%
  filter(genre=="Comedy") %>%
  select(-genre, -c(gross:reviews)) %>%
  mutate(dur_hour = round(duration/60,1))
```

Z kolei funkcja `transmute()` tworzy zbiór w którym jest tylko nowo utworzona kolumna:

```
komedie_t <- movies %>%
  filter(genre=="Comedy") %>%
  select(-genre, -c(gross:reviews)) %>%
  transmute(dur_hour = round(duration/60,1))
```

## 3.6 Zmiana nazwy zmiennej

Do zmiany nazw zmiennych służy funkcja `rename()`. Najpierw podajemy nazwę nowej zmiennej, a po znaku równości starą nazwę:

```
bank <- bank %>%
  rename(karta=kredyt)
```

Zmiany nazwy można także dokonać z wykorzystaniem funkcji `select`:

```
bank_nowy <- bank %>%
  select(lokata=wynik)
```

W takim przypadku trzeba jednak pamiętać o wypisaniu wszystkich zmiennych, które mają się znaleźć w zbiorze wynikowym.

## 3.7 Podsumowanie danych

Funkcja `summarise()` służy do podsumowań danych w formie zagregowanej:

```
bank %>%
  summarise(saldo_srednia=mean(saldo),
            saldo_mediana=median(saldo))
```

```
## # A tibble: 1 x 2
##   saldo_srednia saldo_mediana
##           <dbl>         <dbl>
## 1      1362.272           448
```

Podsumowanie danych ma najwięcej sensu w połączniu z funkcją grupującą.

## 3.8 Grupowanie

Do grupowania obserwacji służy funkcja `group_by()`. Zobaczmy jak wyglądają statystyki salda w poszczególnych grupach wykształcenia:

```
bank %>%
  group_by(wykszt) %>%
  summarise(saldo_srednia=mean(saldo),
            saldo_mediana=median(saldo))
```

```
## # A tibble: 4 x 3
##       wykszt saldo_srednia saldo_mediana
##        <chr>         <dbl>         <dbl>
## 1 podstawowe      1250.950           403
## 2    srednie      1154.881           392
## 3     wyzsze      1758.416           577
## 4       <NA>      1526.754           568
```

Po przecinku w funkcji `group_by()` można wskazać kolejne zmienne grupujące:

```
bank %>%
  group_by(wykszt, hipoteka) %>%
  summarise(saldo_srednia=mean(saldo),
            saldo_mediana=median(saldo))
```

```
## # A tibble: 8 x 4
## # Groups:   wykszt [?]
##       wykszt hipoteka saldo_srednia saldo_mediana
##        <chr>    <chr>         <dbl>         <dbl>
## 1 podstawowe      nie      1571.420         521.0
## 2 podstawowe      tak      1007.594         343.5
## 3    srednie      nie      1340.130         416.5
## 4    srednie      tak      1033.950         380.0
## 5     wyzsze      nie      1919.123         618.0
## 6     wyzsze      tak      1583.978         543.0
## 7       <NA>      nie      1779.766         679.0
## 8       <NA>      tak      1206.788         442.0
```

Przydatna jest także funkcja `n()`, która nie przyjmuje żadnego argumentu i zwraca liczebność zbioru bądź grupy.

```
bank %>%
  group_by(wykszt) %>%
  summarise(liczebnosc=n(),
            saldo_srednia=mean(saldo),
            saldo_mediana=median(saldo))
```

```
## # A tibble: 4 x 4
##       wykszt liczebnosc saldo_srednia saldo_mediana
##        <chr>      <int>         <dbl>         <dbl>
## 1 podstawowe       6851      1250.950           403
## 2    srednie      23202      1154.881           392
## 3     wyzsze      13301      1758.416           577
## 4       <NA>       1857      1526.754           568
```

Jeżeli chcemy tylko wyznaczyć liczebności grup to możemy skorzystać z funkcji `count()`:

```
bank %>%
  group_by(wykszt) %>%
  count()
```

```
## # A tibble: 4 x 2
## # Groups:   wykszt [4]
```

```
##       wykszt     n
##        <chr> <int>
## 1 podstawowe  6851
## 2    srednie 23202
## 3     wyzsze 13301
## 4       <NA>  1857
```

Jedną z kategorii zmiennej wykształcenie jest brak danych (`NA`). Zamienimy tą wartość na kategorię *nieustalone* z wykorzystaniem funkcji `mutate()` oraz `if_else()`. Funkcja `if_else()` przyjmuje trzy argumenty - pierwszy (`condition =`) to warunek, który jest weryfikowany, następnie podajemy wartość, która ma być wprowadzona w przypadku spełnienia warunku (`true =`), a na końcu wartość dla niespełnionego warunku (`false =`). Jest to odpowiednik funkcji `JEŻELI` z Excela.

W omawianym przykładzie warunkiem jest sprawdzenie czy wartości zmiennej `wykszt` są równe `NA`. Jeśli tak to na ich miejsce wprowadzany jest tekst *nieustalone*, a w przeciwnym przypadku pozostaje oryginalna wartość.

```
bank %>%
  mutate(wykszt=if_else(is.na(wykszt), "nieustalone", wykszt)) %>%
  group_by(wykszt) %>%
  count()
```

```
## # A tibble: 4 x 2
## # Groups:   wykszt [4]
##        wykszt     n
##         <chr> <int>
## 1 nieustalone  1857
## 2  podstawowe  6851
## 3     srednie 23202
## 4      wyzsze 13301
```

## 3.9 Sortowanie

Sortowanie jest możliwe z wykorzystaniem funkcji `arrange()`. Jako argument podajemy zmienną według, której chcemy posortować zbiór. Domyślne zbiór sortowany jest rosnąco - od wartości najmniejszych do największych:

```
bank_sort <- bank %>%
  arrange(saldo)
```

Zmiana kierunku sortowania jest możliwa po zastosowaniu funkcji `desc()`:

```
bank_sort <- bank %>%
  arrange(desc(saldo))
```

Sortowanie możemy także zastosować do wyników podsumowania danych:

```
bank %>%
  group_by(wykszt) %>%
  summarise(liczebnosc=n(),
            saldo_srednia=mean(saldo),
            saldo_mediana=median(saldo)) %>%
  arrange(saldo_srednia)
```

```
## # A tibble: 4 x 4
##       wykszt liczebnosc saldo_srednia saldo_mediana
```

```
##        <chr>      <int>         <dbl>         <dbl>
## 1    srednie      23202      1154.881           392
## 2 podstawowe       6851      1250.950           403
## 3       <NA>       1857      1526.754           568
## 4     wyzsze      13301      1758.416           577
```

## 3.10 Łączenie zbiorów

W celu zaprezentowania funkcji łączących dane przygotujemy kilka zbiorów pomocniczych:

```
praca_czas <- bank %>%
  group_by(praca) %>%
  summarise(sr_czas=mean(czas))

praca_saldo <- bank %>%
  group_by(praca) %>%
  summarise(sr_saldo=mean(saldo))

zawod_saldo <- bank %>%
  rename(zawod=praca) %>%
  group_by(zawod) %>%
  summarise(sr_saldo=mean(saldo))
```

Do łączenia dwóch zbiorów danych służy funkcja `inner_join()`, która jako argumenty przyjmuje nazwy zbiorów danych oraz klucz łączenia. Jeśli w obu zbiorach występują kolumny o takich samych nazwach to zostaną potraktowane jako klucz łączenia:

```
praca_czas_saldo <- inner_join(praca_czas, praca_saldo)
```

```
## Joining, by = "praca"
```

Jeśli takie kolumny nie będą istniały to wywołanie funkcji zwróci błąd:

```
praca_czas_saldo <- inner_join(praca_czas, zawod_saldo)
```

```
## Error: `by` required, because the data sources have no common variables
```

W takich przypadku należy wskazać klucz połączenia w postaci `by = c("id1"="id2")`:

```
praca_czas_saldo <- inner_join(praca_czas, zawod_saldo, by=c("praca"="zawod"))
```

Jeśli w jednym ze zbiorów nie ma wszystkich identyfikatorów, które znajdują się w drugim zbiorze to zastosowanie funkcji `inner_join()` będzie skutkowało zbiorem, w którym znajdą się tylko te obserwacje, które udało się połączyć.

```
praca_saldo_1500 <- praca_saldo %>%
  filter(sr_saldo > 1500)

inner_join(praca_czas, praca_saldo_1500, by="praca")
```

```
## # A tibble: 6 x 3
##   praca  sr_czas sr_saldo
##   <dbl>    <dbl>    <dbl>
## 1     2 288.5434 1521.746
## 2     3 253.9958 1763.617
## 3     5 256.3093 1521.470
## 4     7 268.1571 1647.971
```

```
## 5     8 287.3613 1984.215
## 6    NA 237.6111 1772.358
```

Jeśli chcemy pozostawić niedopasowane obserwacje to należy wykorzystać jedną z funkcji - `left_join()` lub `right_join()` w zależności od tego dla którego zbioru chcemy pozostawić wszystkie informacje.

```
left_join(praca_czas, praca_saldo_1500, by="praca")
```

```
## # A tibble: 11 x 3
##    praca  sr_czas sr_saldo
##    <dbl>    <dbl>    <dbl>
##  1     1 246.8967       NA
##  2     2 288.5434 1521.746
##  3     3 253.9958 1763.617
##  4     4 245.8250       NA
##  5     5 256.3093 1521.470
##  6     6 262.9016       NA
##  7     7 268.1571 1647.971
##  8     8 287.3613 1984.215
##  9     9 252.9050       NA
## 10    10 256.9863       NA
## 11    NA 237.6111 1772.358
```

## 3.11 Szeroka i wąska reprezentacja danych

Do wyjaśnienia kwestii szerokiej i wąskiej reprezentacji danych posłużymy się danymi z GUS dotyczącymi przeciętnego miesięcznego spożycie wybranych artykułów żywnościowych na 1 osobę w 2016 roku - plik.

```
spozycie <- read_xlsx("data/spozycie.xlsx")
```

Taka tabela jest przykładem szerokiej reprezentacji danych. Z kolei w niektórych sytuacjach wygodnie jest korzystać z wąskiej reprezentacji danych, a niektóre pakiety wręcz wymagają takich zbiorów wejściowych.

Do transformacji danych z reprezentacji szerokiej na wąską służy funkcja `gather()` (pol. *gromadzić*). Kluczowe są w niej dwa argumenty - pierwszy (`key`) określa nazwę nowej kolumny, która będzie zawierała nazwy zmiennych, a drugi (`value`) określa nazwę nowej kolumny, która będzie zawierała wartości zmiennych. Jako kolejne argumenty podaje się nazwy kolumn, które mają być transformowane lub nazwy kolumn ze znakiem minus `-`, które nie mają być transformowane.

```
spozycie_waskie <- spozycie %>%
  gather(artykul, spozycie, mieso, owoce, warzywa)

# spozycie_waskie <- spozycie %>%
#   gather(artykul, spozycie, -kod, -nazwa)
```

W takiej formie łatwiej podsumować dane:

```
spozycie_waskie %>%
  group_by(artykul) %>%
  summarise(sr_spozycie=mean(spozycie))
```

```
## # A tibble: 3 x 2
##   artykul sr_spozycie
##     <chr>       <dbl>
## 1   mieso    5.468750
## 2   owoce    3.651875
```

```
## 3 warzywa    8.852500
```

W porównaniu do szerokiej reprezentacji danych:

```
spozycie %>%
  summarise(sr_spozycie_mieso=mean(mieso),
            sr_spozycie_owoce=mean(owoce),
            sr_spozycie_warzywa=mean(warzywa))
```

```
## # A tibble: 1 x 3
##   sr_spozycie_mieso sr_spozycie_owoce sr_spozycie_warzywa
##               <dbl>             <dbl>               <dbl>
## 1           5.46875          3.651875              8.8525
```

Transformacja z wąskiej do szerokiej reprezentacji danych jest możliwa z zastosowaniem funkcji `spread()` (pol. *rozprzestrzeniać*). W przypadku tej funkcji niezbędne są dwa argumenty - pierwszy (`key`) wskazuje kolumnę zawierającą nazwy dla nowych zmiennych, a drugi argument (`value`) wskazuje kolumnę zawierającą wartości dla nowych zmiennych.

```
spozycie_szerokie <- spozycie_waskie %>%
  spread(artykul, spozycie)
```

## 3.12 Eksport danych

Zapis zbioru danych do zewnętrznego pliku jest możliwy z wykorzystaniem funkcji `write.table()`. Jako argumenty tej funkcji określamy: zbiór danych (`x`), docelowe miejsce na dysku i nazwę pliku (`file`), separator kolumn (`sep`), separator miejsc dziesiętnych (`dec`) oraz argument `row.names = FALSE`, dzięki któremu unikniemy dodatkowych numerów wierszy.

```
write.table(spozycie_waskie, file = "data/spozycie_w.csv", sep=";", dec=",", row.names=F)
```

Taki plik jest plikiem csv, który możemy otworzyć w Excelu i zapisać go z rozszerzeniem .xlsx. Teoretycznie istnieje pakiet `xlsx`, który umożliwia zapisywanie zbiorów od razu do Excela, ale działa w oparciu o Javę, co bywa problematyczne.

## 3.13 Zadania

Na podstawie zbioru rossmann odpowiedź na pytania:

1. Ile było sklepów o asortymencie rozszerzonym w dniu 25-02-2014?
2. W jaki dzień tygodnia średnia liczba klientów była największa w sklepie nr 101?
3. Sklep jakiego typu charakteryzuje się największą medianą sprzedaży?
4. Czy w ciągu roku odległość do najbliższego sklepu konkurencji zmieniła się dla jakiegokolwiek sklepu Rossmann?

## 3.14 Case study

Rozważmy sklepy Rossmann, które w 2014 roku były otwarte powyżej 300 dni w roku. Czy średnia sprzedaż wyrażona w zł w sklepach Rossmann różni się statystycznie pomiędzy grupami zdefiniowanymi przez asortyment sklepu? Dane na temat średniego kursu miesięcznego euro pobierz ze stron NBP.

# Chapter 4

# Wizualizacja danych

## 4.1 Wbudowane funkcje

## 4.2 Pakiet ggplot2

### 4.2.1 Analiza rozkładu

Będziemy działać na zbiorze dotyczącym klientów banku

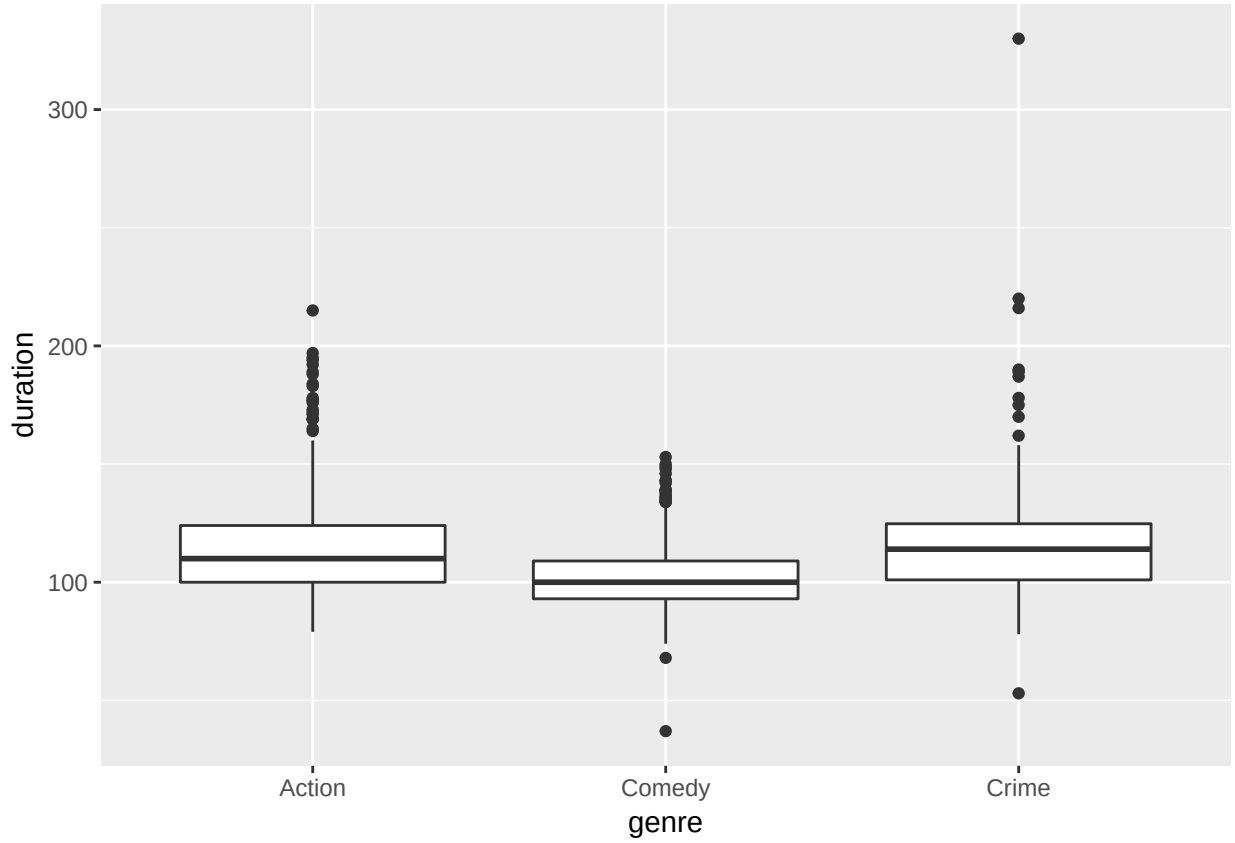
```
library(tidyverse)
library(readxl)

movies <- read_csv("data/movies.csv")
```

```
## Parsed with column specification:
## cols(
##   title = col_character(),
##   genre = col_character(),
##   director = col_character(),
##   year = col_integer(),
##   duration = col_integer(),
##   gross = col_integer(),
##   budget = col_integer(),
##   cast_facebook_likes = col_integer(),
##   votes = col_integer(),
##   reviews = col_integer(),
##   rating = col_double()
## )
```

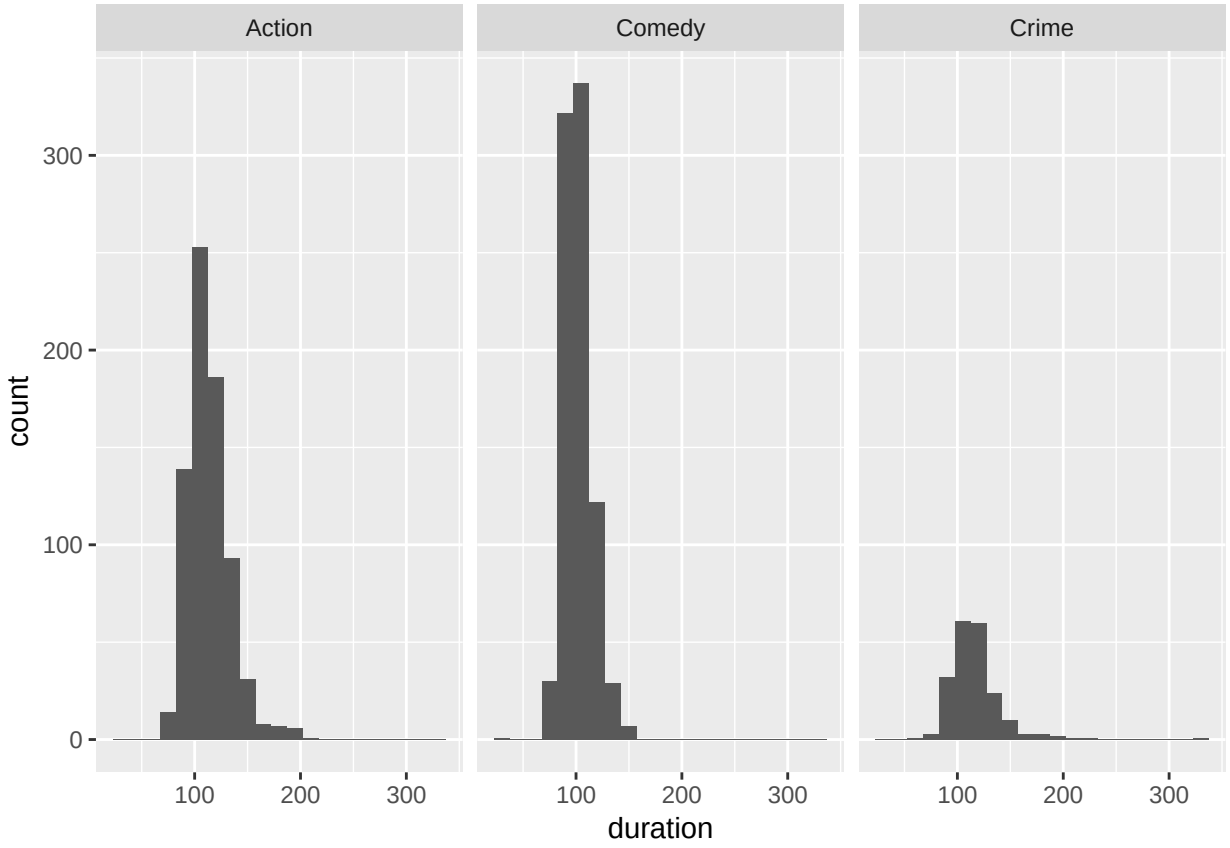
Pierwszym sposobem analizy rozkładu jest wykres pudełkowy

```
movies %>%
  filter(genre %in% c("Action", "Comedy", "Crime")) %>%
  ggplot(., aes(y=duration, x=genre)) +
  geom_boxplot()
```




Te same dane można przedstawić także na histogramie.

```
movies %>%
  filter(genre %in% c("Action", "Comedy", "Crime")) %>%
  ggplot(., aes(x=duration)) +
  geom_histogram(binwidth=15) +
  facet_wrap(~ genre)
```


Action
Comedy
Crime
count
duration
0
100
200
300
100
200
300

# Chapter 5

# Programowanie w R

## 5.1 Funkcje

## 5.2 Pętle

## 5.3 Instrukcje warunkowe