Krótkie wprowadzenie do pakietu R

Statystyka matematyczna i ekonometria

dr inż. Robert Kapłon

ver. 0.9
(kompilacja: 26 września 2014)

Spis treści

Wstęp

Niniejsze opracowanie ma być łagodnym wprowadzeniem do pakietu **R**. Nie jest to wprowadzenie, które nawet w najmniejszym stopniu aspiruje do miana kompletnego. Wiele istotnych treści zostało pominiętych, ze wszech miar celowo, włączają w to nawet podstawowe zagadnienia. Chcę uniknąć efektu przytłoczenia, a jednocześnie ułatwić start z tym fantastycznym środowiskiem. Zainteresowanych poszerzeniem wiedzy odsyłam do bardzo bogatej literatury. Przykładowo z wykazem książek można zapoznać się pod adresem: `http://www.r-project.org/doc/bib/R-books.html`; warto też zwrócić uwagę na pozycje bibliograficzne zamieszczone na końcu opracowania. Tym szczególnie dociekliwym, chcącym poznać fundamenty język, a nie tylko gotowe „przepisy", polecam wspaniałą książkę: *Programowanie w R* autorstwa Pana Marka Gągolewskiego (4).

W wielu miejscach, a szczególnie przy wizualizacji danych, zamieszczam przykłady różniące się nieznacznie. Ich analiza pozwala szybko dostrzec różnice między użytymi funkcjami i argumentami. Czasami takie podejście jest lepsze, niż szczegółowy opis. Dlatego wykresów nie należy traktować jako tych, które zostały przygotowane zgodnie ze sztuką. Czasami zdarza się, że wyglądają dziwacznie.

Opanowanie umiejętności na poziomie tego wprowadzenia jest wystarczający do wykonania nawet złożonych analiz. Pamiętać jednak należy, że samo czytanie — bez aktywnego, równoczesnego używania programu — jest niewystarczające do zdobycia biegłości w posługiwaniu się **R**. W kontekście tego zachęcam do przepisywania zamieszczonych tutaj fragmentów programów, a nie ich kopiowania i wklejania. Wpisując kilka czy kilkanaście razy tą samą funkcję, po prostu ją zapamiętujemy. Również polecam przeprowadzanie eksperymentów, gdy podczas lektury nasuwają się wątpliwości. Bardzo często można sobie samemu odpowiedzieć na nurtujące pytania zmieniając — nawet w niewielkim stopniu — zamieszczony tu kod, i sprawdzając, co się stanie.

W opracowaniu wykorzystuję różne zbiory danych, które można ściągnąć, z lokalizacji podanej na wykładzie.

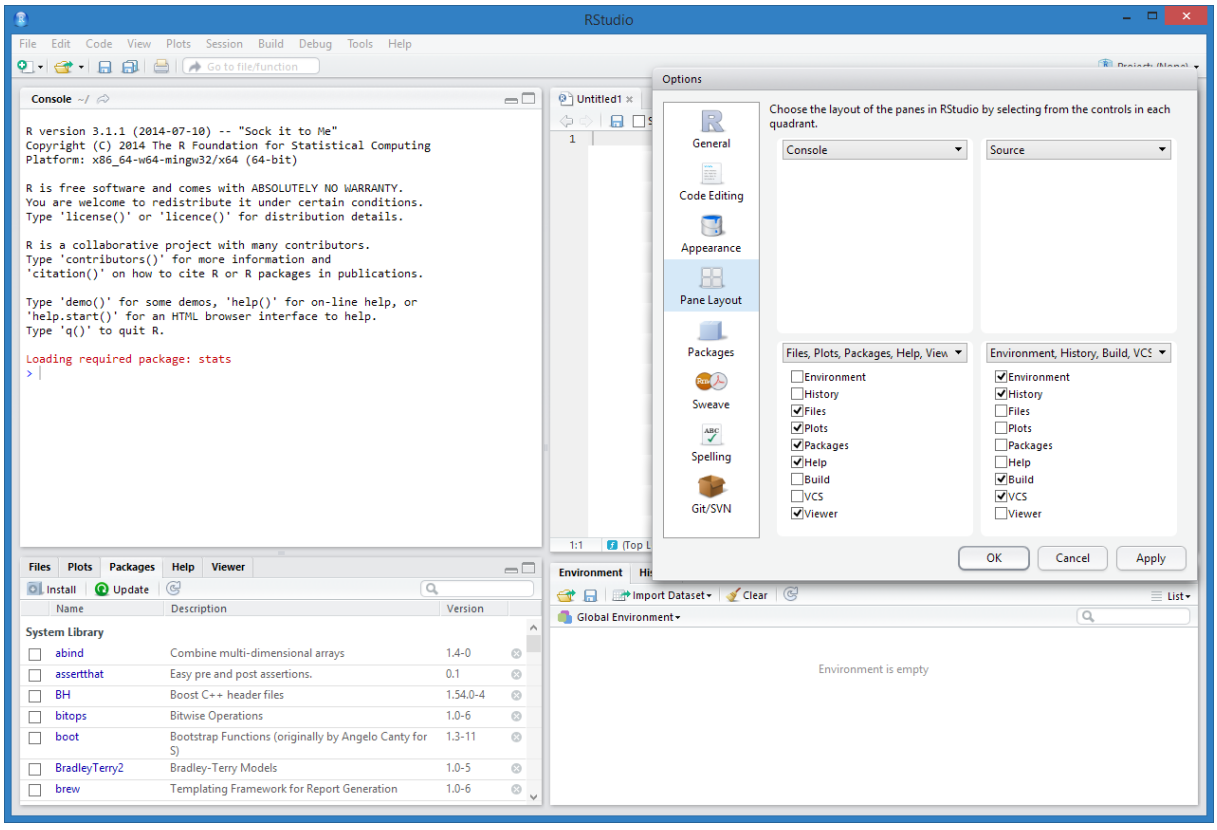
Rozdział 1

Instalacja i konfiguracja

Na stronie **R**-projektu, z menu po lewej stronie, wybieramy CRAN a następnie lokalizację, z której chcemy ściągnąć plik. Teoretycznie każdy URL powinien zawierać tą samą wersję programu. Pamiętajmy o platformie, na której **R** będzie instalowany. Po wyborze np. *Windows*, wskazujemy *base* i pobieramy plik. Wybierzmy pełną instalację z domyślnymi ustawieniami zmieniając tylko na pojedyncze okno (*Window SDI* a nie domyślne *MDI*). Nie jest to kluczowe ustawienie, więc w razie trudności można je pominąć.

Od pewnego czasu pewne komunikaty w konsoli **R** są w j. polskim, z kolei pomoc, dokumentacja, tutoriale itp. są w j. angielskim. Taka dwujęzyczność przeszkadza, dlatego będziemy pracować na wersji angielskiej. Aby dokonać zmiany na ten język, należy edytować w notatniku plik `Rconsole` — wystarczy najechać na plik i prawym przyciskiem myszki wybrać: otwórz za pomocą, wskazując np. na notatnik. Plik znajduje się w katalogu `etc`. U mnie pełna ścieżka to: `C:\R\R-3.1.1\etc\`. Gdy już otworzymy plik, należy odszukać wiersz odnoszący się do definicji języka i dopisać `en`; tak powinien wyglądać wiersz: `language = en`.

Do tworzenia i edytowania skryptów **R** wykorzystamy darmowe, zintegrowane środowisko RStudio.



Całe okno RStudio podzielone jest na 4 panele główne, które można konfigurować, wybierając z menu: *Tools* i *Global Options*. W ten sposób wywołamy okno *Options*, widoczne na powyższym rysunku.

W powyższej konfiguracji prawy górny panel służy do pisania kodu programu. Linijkę kod możemy przekazać do konsoli **R** (znajdującej się w lewym górnym panelu) za pomocą kombinacji klawiszy: `CTRL+Enter`. Jeśli chcemy, aby tych linii było więcej, wtedy należy je zaznaczyć i użyć tego skrótu. Z innymi, dostępnymi skrótami klawiaturowymi można się zapoznać wybierając z menu `Help` i `Keyboard Shortcuts`.

Konsola **R** pozwala na interaktywną pracę. Wpiszmy w niej `2+2` i naciśnijmy Enter. Nastąpi natychmiastowa interpretacja, a wynik pojawi się poniżej. Jeśli spróbujmy wpisać: `#2+2`, wtedy nic się nie stanie, gdyż każdy tekst występujący po znaku # jest traktowany jako komentarz. Dobrym zwyczajem jest komentowanie kodu.

Tryb interaktywny jest użyteczny, gdy chcemy otrzymać natychmiastowy wynik lub sprawdzić, czy składnia której używamy, jest poprawna. Przykładowo, pamiętamy z matematyki, że funkcja ln(23) oblicza wartość logarytmu naturalnego z liczby 23. Wpisując to w konsoli, **R** zgłosi błąd. Okazuje się, że funkcja $\ln(x)$ w **R** ma postać $\log(x)$ — polecam sprawdzić.

Jeśli chcielibyśmy się dowiedzieć czegoś więcej na temat tej funkcji logarytmicznej, wpiszmy w konsoli `?log`. Zawsze, kiedy użyjemy pytajnika, podając po nim nazwę funkcji, zgłosi się okno pomocy **R**. Jeśli nie znamy dokładnej nazwy funkcji, a chcemy wyszukać wszystko to, co dotyczy pewnej frazy (musimy użyć cudzysłowów) używamy podwójnego pytajnika `??`. Polecam sprawdzić: `??log`, `??"log normal"`.

Pomocy można również poszukiwać w dokumentacji i opracowaniach na stronie **R**:

- Manuale: `http://cran.r-project.org/manuals.html`
- Pozostałe dokumenty: `http://cran.r-project.org/other-docs.html`
- Tytuły książek: `http://www.r-project.org/doc/bib/R-books.html`

Bardzo mocą stroną **R** są pakiety, tworzone przez grono entuzjastów i miłośników zarówno samego języka jak i również szeroko pojętej analizy danych. Oferują one funkcjonalność niespotykaną w innych pakietach statycznych. Okazuje się, że wiele pomysłów — które przychodzą nam do głowy — na analizę danych zostało już zaimplementowanych w postaci pakietów, które w liczbie prawie 6000 są dostępne w repozytorium CRAN na stronie **R**. Aby móc używać takiego pakietu należy go najpierw zainstalować wywołując funkcję `install.packages("tutaj_nazwa_pakietu")`. Tę czynność wykonujemy tylko raz. Jeśli takiego pakietu chcemy użyć, wtedy przy każdym uruchomieniu programu RStudio trzeba wywołać funkcję: `library("tutaj_nazwa_pakietu")`, aby wczytać pakiet.

```
> install.packages("car") #instalujemy tylko raz
> library(car) #za każdym razem (w nowej sesji R), gdy chcemy użyć
```

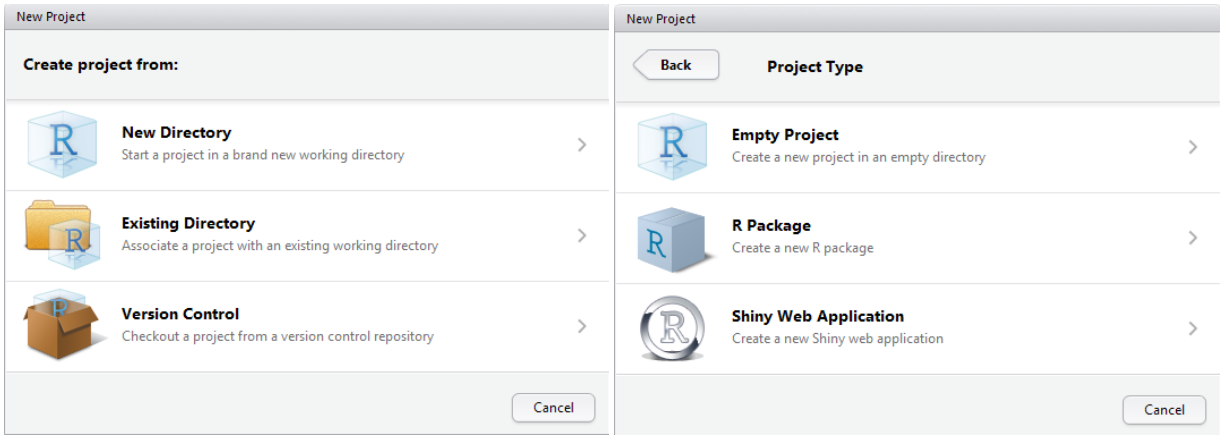
W zakresie pakietów użyteczne mogą być jeszcze dwie funkcje. Pierwsza z nich pokazuje listę wszystkich zainstalowanych pakietów na komputerze. Druga z kolei wyświetla nazwy tych pakietów, które zostały już wczytane do **R**.

```
> .packages(all.available = TRUE) #ponieważ jest ich dużo, nie wyświetlę ich
```

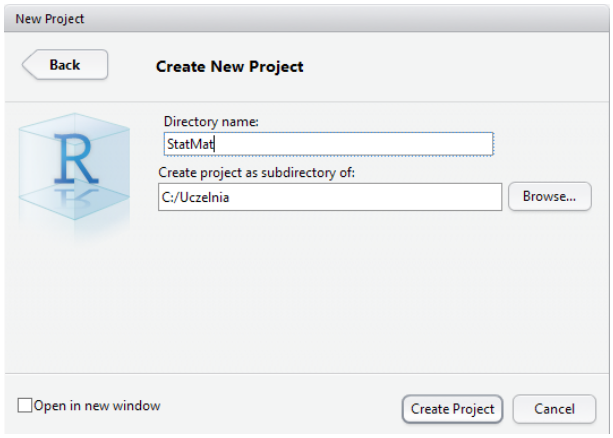
```
> (.packages()) #pokazuje, które z zainstalowanych pakietów zastały już wczytane

 [1] "stringr"      "tools"        "knitr"        "graphics"     "grDevices"
 [6] "utils"        "datasets"     "dplyr"        "ggplot2"      "methods"
[11] "stats"        "RColorBrewer" "base"
```

1.1. Tworzenie projektu

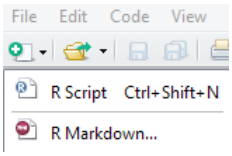
Przy rozbudowanych programach warto utworzyć projekt. Dla przykładu stworzymy taki projekt dla przedmiotu, nazwijmy go: *StatMat*. W pierwszym kroku z menu wybieramy *File* i *New Project...*. Gdy pojawi się okno (zob. poniżej) wybieramy: *New Directory*, a następnie *Empty Project*.



Ostatni krok (poniższe okno) polega na podaniu nazwy projektu (tutaj *StatMat*) oraz katalogu, w którym będzie on zapisany (tutaj dla przykładu `C:/Uczelnia`).



Od teraz możemy całą zawartość katalogu *StatMat* przenosić między komputerami. Aby otworzyć projekt, wystarczy kliknąć na plik `StatMat.Rproj`, znajdujący się w tym katalogu. W ramach tego projektu powstanie zapewne wiele plików z napisanymi programami (skryptami). Aby utworzyć nowy skrypt klikamy na ikonkę zielonego plusa i wybieramy `R Script`.



Jeśli chcemy mieć osobny plik, dla każdego rozdziału, z rozwiązanymi zadaniami wtedy nazwa pliku może wyglądać tak: `zad_rozdz1.R`. Nie zapomnijmy o rozszerzeniu po kropce i dużej literze `R`.

Rozdział 2

Struktury danych

2.1. Wektory

Przez wektor będziemy rozumieć ciąg elementów tego samego typu. A więc elementami mogą być albo liczby (naturalne, całkowite, rzeczywiste, zespolone) albo wartości logiczne albo znaki (łańcuchy znaków). Typów nie można mieszać — naruszenie tej zasady zawsze skutkuje uzgadanianiem jednego typu przez **R** (tzw. koercja). Oto kilka przykładów z uwzględnieniem dwóch stałych logicznych występujących w **R** (`TRUE`, `FALSE`):

$$\mathbf{x} = \begin{bmatrix} 1 \\ -7 \\ 5 \\ 3 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} -1.37 \\ -22.1 \\ 4.6 \\ 3 \end{bmatrix}, \quad \mathbf{z} = \begin{bmatrix} \text{TRUE} \\ \text{FALSE} \\ \text{FALSE} \\ \text{TRUE} \end{bmatrix}, \quad \mathbf{v} = \begin{bmatrix} a \\ abc \\ d \\ kot \end{bmatrix}.$$

Warto odnotować, że mówiąc o wektorach, zawsze będziemy mieli na myśli wektory kolumnowe, a ich wymiar będziemy oznaczać w postaci: $n \times 1$, gdzie n odpowiada liczbie elementów. Dodatkowo liczba elementów jest równa długości wektora. W powyższym przykładzie $n = 4$.

Tworzenie wektorów

Wektory w **R** możemy tworzyć wykorzystując:

- funkcję `c()`, (*combine*, czyli połącz);
- operator `:`, tworzący ciąg arytmetyczny o różnicy 1;
- funkcję `seq()`, (*sequence*, czyli ciąg) tworzy ciąg arytmetyczny ;
- funkcję `rep()`, (*replicate*, czyli powtórz) powtarza elementy zadaną liczbę razy.

Pomoc na temat składni powyższych funkcji uzyskamy wpisując w konsoli **R** znak zapytania i nazwę funkcji, np: `?seq`. Prześledźmy na przykładzie ich użycie.

```
> ## funkcja c() i wektory rożnych typów
> c(1, 5, 6, -2.34) #typ numeryczny

[1]  1.00  5.00  6.00 -2.34

> c("zdecydowanie", "raczej nie", "trudno sie zdecydowac") #typ znakowy (łańcuch znaków)

[1] "zdecydowanie"          "raczej nie"            "trudno sie zdecydowac"

> c(TRUE, FALSE, FALSE) #typ logiczny

[1]  TRUE FALSE FALSE

> c(1, 7, "zarobki") # niedozwolone mieszanie typów: R uzgodni typ (będzie to znakowy)

[1] "1"       "7"       "zarobki"
```

```
> ## operator :
> 3:10 # utwórz wektor od 3 do 10

[1]  3  4  5  6  7  8  9 10

> c(3:10) #to samo co wyżej

[1]  3  4  5  6  7  8  9 10

> 7:2

[1] 7 6 5 4 3 2

> c(3:10, 4:2, -3)

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

```
> ## funkcja seq() z różnymi argumentami
> ## seq(from = ..., to = ..., by = ..., length.out = ...)
> ## by  - co ile przyrost; length.out - jak długi wektor
> seq(from=1, to=10, by=0.4) # liczby od 1 do 10 z przyrostem 0.4

 [1] 1.0 1.4 1.8 2.2 2.6 3.0 3.4 3.8 4.2 4.6 5.0 5.4 5.8 6.2 6.6 7.0 7.4 7.8 8.2 8.6 9.0
[22] 9.4 9.8

> seq(1, 10, 0.4) #to samo co wyżej; argumenty można pominąć, gdy wpisujemy w kolejności

 [1] 1.0 1.4 1.8 2.2 2.6 3.0 3.4 3.8 4.2 4.6 5.0 5.4 5.8 6.2 6.6 7.0 7.4 7.8 8.2 8.6 9.0
[22] 9.4 9.8

> seq(1, 10, length.out=30) #R ustala przyrost, aby długość=30

 [1]  1.00000  1.31034  1.62069  1.93103  2.24138  2.55172  2.86207  3.17241  3.48276
[10]  3.79310  4.10345  4.41379  4.72414  5.03448  5.34483  5.65517  5.96552  6.27586
[19]  6.58621  6.89655  7.20690  7.51724  7.82759  8.13793  8.44828  8.75862  9.06897
[28]  9.37931  9.68966 10.00000
```

```
> ## funkcja rep() z różnymi argumentami
> ## rep(x, times = ..., length.out = ..., each = ...)
> rep(c(1, 5, 3), times=5) # powtórz 5 razy wektor c(1, 3, 5)

 [1] 1 5 3 1 5 3 1 5 3 1 5 3 1 5 3

> rep(c(1, 5, 3), each=5) # powtórz 5 razy każdy element wektora c(1, 5, 3)

 [1] 1 1 1 1 1 5 5 5 5 5 3 3 3 3 3

> rep(c(1, 5, 3), times=c(2, 3, 4)) #powtórz: 1 - dwa razy, 5 - trzy razy, 3 - cztery razy

[1] 1 1 5 5 5 3 3 3 3

> rep(c("nie", "powtarzaj", "sie"), times=4)

 [1] "nie"       "powtarzaj" "sie"       "nie"       "powtarzaj" "sie"       "nie"
 [8] "powtarzaj" "sie"       "nie"       "powtarzaj" "sie"
```

Operator przypisania

Jeśli chcemy zapisać informację w pamięci komputera — aby później ją wykorzystać i poddać dalszemu przetwarzaniu — to należy utworzyć obiekt nazwany, wykorzystując operator przypisania <-. Choć można też użyć znaku = to z pewnych względów nie zalecam.

W **R** nazwy mogą składać się z ciągu liter, cyfr, kropki, podkreślenia. Nie można nazw zaczynać od liter, używać znaków specjalnych (np. %, #) oraz słów kluczowych (np. `if`, `else`, `TRUE`, `FALSE`, `NA`). Przykładem poprawnych nazw w **R** są: `grupaWiek`, `grupa.wiek`, `grupa_wiek`, `GrupaWiek`. Ponieważ wielkość liter w **R** jest istotna, dlatego pierwsza i ostatnia nazwa nie są traktowane identycznie.

```
> x <- c(1, 3, -3.45, 4.123) #przypisz wartości wektora do x (utwórz obiekt x)
> plec <- c("k", "m", "m", "m", "k")
> stawkaGodz <- seq(10, 20, length.out=15)
> ## Wpisując powyższe w konsoli, nie widzimy efektu.
> ## Ale R ma już w pamięci wektory: x i plec
> ## wpisując w konsoli x bądź plec zobaczymy elementy
> x

[1]  1.000  3.000 -3.450  4.123

> plec

[1] "k" "m" "m" "m" "k"

> stawkaGodz

 [1] 10.0000 10.7143 11.4286 12.1429 12.8571 13.5714 14.2857 15.0000 15.7143 16.4286
[11] 17.1429 17.8571 18.5714 19.2857 20.0000

> ## Kilka operacji na wektorach z wykorzystaniem wbudowanych funkcji
> length(stawkaGodz) # długość wektora stawkaGodz

[1] 15

> typeof(plec) #jakiego typu jest wektor pleć (oczywiście znakowego)

[1] "character"

> stawkaGodz <- plec #podstaw plec za  stawkaGodz
> stawkaGodz #powyższa operacja nadpisała wcześniej zdefiniowane wartości

[1] "k" "m" "m" "m" "k"
```

Wartości i symbole specjalne

- `Inf` — wartość nieskończona
- `NaN` (*not a number*) — wyrażenie nieoznaczone (np. konsekwencja wykonania działania $\frac{0}{0}$)
- `NA` (*not available*) — jeśli mamy do czynienia z brakującymi danymi, wtedy ten fakt zostanie odnotowany przez `NA`
- `NULL` — typ pusty, traktujemy jako element/zbiór pusty.

```
> log(0) #oblicz logarytm naturalny z 0

[1] -Inf

> sqrt(-1) #pierwiastek kwadratowy z liczby ujemnej (dla liczb rzeczywistych nie istnieje)
```

```
Warning: NaNs produced

[1] NaN

> c(3, 5, 9, NA) #ostatni element wektora traktowany jako brak danej

[1]  3  5  9 NA

> c(5, 3, NULL, 8) #wektor ma 3 elementy (pusty z definicji pominięty)

[1] 5 3 8
```

Podstawowe operacje na wektorach

W **R** mamy do dyspozycji następujące operatory:

- operatory arytmetyczne: `+`, `-`, `*`, `/`, `^`, (dodawanie, odejmowanie, mnożenie, dzielenie, potęgowanie); przykłady:

  `x+y`, `x-y`, `,x*y`, `x/y`, `x^y`

- operatory logiczne: `!`, `|`, `&` (negacja, alternatywa, koniunkcja); alternatywa i koniunkcja dla skalarów to odpowiednio symbole `||`, `&&`; przykłady

  `!x`, `x|y`, `x&y`, gdy skalary: `x||y`, `x&&y`

- operatory relacyjne: `>`, `<`, `>=`, `<=`, `==`, `!=` (większy, mniejszy, większy bądź równy, mniejszy bądź równy, równy, różny); przykłady

  `x > y`, `x < y`, `x >= y`, `x <= y`, `x == y`, `x != y`

Podstawowe działania na wektorach odbywają się z użyciem operatorów arytmetycznych. Jeśli wektory są takiej samej długości, wtedy działania wykonywane są element po elemencie: $[1,4] + [2,5] = [3,9]$. Gdy jeden z nich jest krótszy, wtedy jest on powielany tyle razy, aby zrównał się z długością tego dłuższego. Jest to tzw. ***reguła zawijania*** (*recycling rule*). Przykładowo, jeśli chcemy dodać dwa wektory: $[5,7,3]$ i $[1,3,7,4,9,2]$, wtedy ten pierwszy zostanie powielony dwa razy i po otrzymaniu $[5,7,3,5,7,3]$ dodany do drugiego. Pierwszy i drugi mają długości odpowiednio: 3 i 6. Ponieważ 6 jest wielokrotnością 3, więc **R** po wykonaniu operacji nie wyświetli żadnego komunikatu. Gdyby jednak długość większego nie była wielokrotnością mniejszego, wtedy również zostania zastosowana reguła zawijania (pierwszy zostanie rozszerzony do: $[5,7,3,5,7,3,5]$), ale po wykonaniu działania pojawi się odpowiedni komunikat. Zanim przejdziemy do przykładów, zastanówmy się, w jaki sposób mnożona jest liczba przez wektor. Otóż liczba, to *de facto* wektor z jednym elementem. Dlatego reguła zawijania tutaj też obowiązuje. Przemnożenie liczby 2 przez wektor $[5,7,3]$ to mnożenie dwóch wektorów: $[2,2,2]$ i $[5,7,3]$ element po elemencie.

```
> x <- c(5, 7, 3) # długość 3
> y <- c(1, 3, 7, 4, 9, 2) #długość 6
> x + y #zawijanie bez komunikatu, bo 6/3 jest całkowite

[1]  6 10 10  9 16  5

> y <- c(1, 3, 7, 4, 9, 2, 100) #długość 7
> x + y #zawijanie z ostrzeżeniem

Warning: longer object length is not a multiple of shorter object length

[1]   6  10  10   9  16   5 105

> x + 1000

[1] 1005 1007 1003
```

```
> x <- c(8,  2,  4,  12, 10,  6)
> y <- c(1,  5,  9,  0,  3,  -5)
> z <- c(8, 9, 0)
> x - y #odejmowanie

[1]  7 -3 -5 12  7 11

> x * y #mnożenie

[1]   8  10  36   0  30 -30

> z / x #reguła zawijania (dla którego wektora?)

[1] 1.000000 4.500000 0.000000 0.666667 0.900000 0.000000

> y^2 #podnieś do 2 potęgi

[1]  1 25 81  0  9 25

> z^2 + y #podnieś do 2 potęgi i dodaj y

[1] 65 86  9 64 84 -5
```

Do każdego elementu wektora można się odwołać, wskazując indeks (pozycję) elementu. Numerowanie elementów zaczyna się od 1 (w niektórych językach od 0, np. C++). Jeśli interesuje nas trzeci element wektora x, wystarczy wpisać nazwę wektora, a w nawiasach kwadratowych wskazać pozycję na jakiej występuje element, czyli $x[3]$.

```
> ## Rozważmy krótki przykład: dzienny utarg z wizyt
> stawka <- c(100, 70, 90, 150, 120, 110, 130) #koszt wizyty u specjalisty
> ilePacjent <- c(3, 5, 4, 4, 1, 7, 3) #ilu pacjentów przyjął
> utarg <- stawka * ilePacjent
> utarg

[1] 300 350 360 600 120 770 390

> utarg[3] #wybiera 3 elment

[1] 360

> utarg[3:5] #wybiera elementy od 3 do 5

[1] 360 600 120

> utarg[c(1, 3, 4, 5)] #wybiera wskazane elementy

[1] 300 360 600 120

> utarg[-2] #wybiera wszystkie bez elementu 2

[1] 300 360 600 120 770 390

> utarg[-c(1, 4)] #wybiera wszystkie bez elementu 1 i 4

[1] 350 360 120 770 390

> doktor <- c(1, 3, 4, 7)
> utarg[doktor]

[1] 300 360 600 390

> utarg[c(TRUE, FALSE, TRUE, TRUE, TRUE, FALSE, FALSE)] #wybiera element dla TRUE

[1] 300 360 600 120
```

Zobaczmy jeszcze, jak można wykorzystać operatory logiczne i relacyjne w przykładzie: dzienny utarg z wizyt

```
> ## Mamy wektor utarg
> utarg

[1] 300 350 360 600 120 770 390

> utarg[utarg > 300] #wez utarg powyżej 300

[1] 350 360 600 770 390

> utarg[utarg > 350 | utarg <200] #wez utarg większy od 350 lub mniejszy od 200

[1] 360 600 120 770 390

> utarg[utarg < 350 & utarg >250] #wez utarg między 250 a 350

[1] 300
```

W powyższym przykładzie **R** przetwarza dwuetapowo. Najpierw wyznacza wartość wyrażenia w nawiasie kwadratowym: wynikiem jest wektor logiczny, o czym można się przekonać wpisując te wyrażenia w konsoli:

```
> utarg > 300

[1] FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE

> utarg > 350 | utarg <200

[1] FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE

> utarg < 350 & utarg >250

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

W drugim kroku wybiera tylko te pozycje wektora `utarg`, którym odpowiada wartość `TRUE` (prawda).

Wybrane (wbudowane) funkcje

Poniżej podaję podstawowe funkcje matematyczne, których argumentami mogą być również wektory. W tym wypadku operacje wykonywane są element po elemencie.

Funkcja	**Opis**
`log(x)`	Logarytm naturalny z `x`
`exp(x)`	Liczba e podniesiona do potęgi `x`
`log(x, n)`	Logarytm z `x` przy podstawie `n`
`sqrt(x)`	Pierwiastek kwadratowy z `x`
`factorial(n)`	$n! = 1 \cdot 2 \cdot \ldots \cdot n$
`choose(n, k)`	Symbol Newtona $\frac{n!}{k!(n-k)!}$
`cos(x), sin(x), tan(x)`	Funkcje trygonometryczne
`abs(x)`	Wartość bezwzględna z `x`
`floor(x)`	Największa liczba całkowita $\leqslant$ `x`
`ceiling(x)`	Najmniejsza liczba całkowita $\geqslant$ `x`
`trunc(x)`	Bierze część całkowitą `x`
`round(x, digits=n)`	Zaokrągla `x` do `n` miejsc po przecinku

```
> ##Przykłady wykorzystania funkcji matematycznych dla wektorów
> x <- rnorm(10) # generuje 10 liczb losowych z rozkładu normalnego
> x

 [1]  0.0948455  0.5510664  0.3056478  1.6490266  0.4249424  0.7127905  0.4638044
 [8] -0.2056986  0.3896563 -0.0913869

> round(x, 1) #zaokrąglić do 1 miejsca po przecinku

 [1]  0.1  0.6  0.3  1.6  0.4  0.7  0.5 -0.2  0.4 -0.1

> exp(x) #obliczyć wartość funkcji e w punktach x

 [1] 1.099489 1.735102 1.357504 5.201914 1.529502 2.039675 1.590112 0.814078 1.476473
[10] 0.912664

> abs(x) #obliczyć wartość bezwzględna

 [1] 0.0948455 0.5510664 0.3056478 1.6490266 0.4249424 0.7127905 0.4638044 0.2056986
 [9] 0.3896563 0.0913869

> log(abs(x)) #najpier oblicz wartość bezwzględną, później log. naturalny

 [1] -2.355506 -0.595900 -1.185322  0.500185 -0.855802 -0.338568 -0.768292 -1.581343
 [9] -0.942490 -2.392653
```

Funkcje operujące na wektorach, tzw. funkcje agregujące.

Funkcja	Opis
`length(x)`	Długość (liczba elementów) wektora `x`
`max(x)`	Największa wartość z `x`
`min(x)`	Najmniejsza wartość z `x`
`sum(x)`	Suma wszystkich wartości `x`
`prod(x)`	Iloczyn wszystkich wartości `x`
`range(x)`	Podaje wartość `max(x)` i `min(x)`
`sort(x,decreasing = FALSE)`	Sortuje (rosnąco) wartości `x`; gdy `TRUE` - malejąco
`cumsum(x)`	Skumulowana suma wszystkich wartości `x`
`cumprod(x)`	Skumulowany iloczyn wszystkich wartości `x`
`pmax(x, y, z)`	Zestawia wektory `x,y,z` w kolumny i wybiera największą wartość w każdym wierszu
`pmin(x, y, z)`	Zestawia wektory `x,y,z` w kolumny i wybiera najmniejszą wartość w każdym wierszu
`sample(x, n, replace=TRUE)`	Losowanie `n` elementów wektora `x` ze zwracaniem (`replace =TRUE`) lub bez (`replace=FALSE`)
`which(x)`	Zwraca te indeksy wektora logicznego `x`, które mają wartość `TRUE`, np `which(x == 5)` podaje indeksy wektora `x` równe 5.
`which.max(x), which.min(x)`	Zwraca indeks elementu największego i najmniejszego
`unique(x)`	Usuwa duplikaty wektora `x`. Jeśli $x = [1,3,2,1,3,2,1]$ to `unique(x)` zwróci $[1,3,2]$

```
> ##Przykład: z wektora wartości od 1 do 100 wylosujemy 10 liczb
> set.seed(76) #ustaw ziarno generatora (gwarantuje identyczność losowania)
> los <- sample(1:100, 10, replace=FALSE)
> los
```

```
 [1] 63 73 43 10 45 99 29 68 30 58

> max(los)

[1] 99

> sum(los)

[1] 518

> sort(los) #argument decreasing pominięty, dlatego użyty domyślny FALSE

 [1] 10 29 30 43 45 58 63 68 73 99

> zestawienie <- c(range(los), sum(los), length(los))
> zestawienie

[1]  10  99 518  10

> sum(los)/length(los) #to średnia arytmetyczna

[1] 51.8
```

Prześledźmy działanie funkcji `which()`, którą przyjdzie nam wykorzystywać wiele razy. Jeszcze raz chcę to podkreślić: funkcja zwraca indeksy, pozycję elementów, a nie wartości elementów.

```
> which(los > 65) #które elementy (nr indeksu) są większe od 65

[1] 2 6 8

> which.max(los) #indeks elementu największego (pierwszego napotkanego)

[1] 6

> which.min(los) # indeks elementu najmniejszego (pierwszego napotkanego)

[1] 4
```

Zobaczmy, że indeksom 2, 6, 8 odpowiadają wartości 73, 99, 68, które faktycznie są większe od 65. A jak wyświetlić te szukane wartości? Podejście jest identyczne, jak przy odwoływaniu się do elementów wektora. Należy w nawiasach kwadratowych umieścić wektor, którego wartościami są numery indeksów.

```
> los[which(los > 65)]

[1] 73 99 68
```

2.2. Macierze

Macierz jest tablicą dwuwymiarową, składającą się z elementów rozmieszczonych w wierszach i kolumnach. Poniższa macierz $\mathbf{X}$ składa się z n wierszy i k kolumn

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1k} \\ x_{21} & x_{22} & \dots & x_{2k} \\ \dots & \dots & \dots & \dots \\ x_{n1} & x_{n2} & \dots & x_{nk} \end{bmatrix}$$

Indeksy mówią nam, jaka jest pozycja elementu, np element x_{ij} znajduje się w wierszu i oraz kolumnie j. Taką macierz można również traktować jako zbiór wektorów kolumnowych. W **R** taka macierz jest po prostu wektorem z dodatkowym atrybutem `dim`, mówiącym o wymiarze macierzy. W powyższym przykładzie macierz jest wymiaru $n \times k$, a więc `dim` jest wektorem dwuelementowym `c(n, k)`.

Aby utworzyć macierz w **R**, wykorzystujemy funkcję: `matrix(x, nrow, ncol)`, w której: `x` — wektor, `nrow` — liczba wierszy, `ncol` — liczba kolumn. Zauważmy, że wystarczy podać tylko jeden argument (`nrow` lub `ncol`), gdyż ten brakujący zostanie wyznaczony na podstawie długości wektor. Przykładowo, wektor ma długość 50, a liczba kolumn wynosi 5. Ile mamy wierszy? Oczywiście 50/5 = 10. I jeszcze jedna uwaga: elementy macierzy tworzone są w kolejności kolumnowej. Prześledźmy to na przykładach.

```
> ## Definiowanie macierzy
> (x <- 1:15) #nawias powoduje wyświetlenie wartości x w konsoli

 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15

> matrix(x, nrow=3, ncol=5) #najpierw pierwsza kolumna powstaje, później druga itd.

     [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    7   10   13
[2,]    2    5    8   11   14
[3,]    3    6    9   12   15

> matrix(x, nrow=3) #liczba wierszy jest wystarczająca do utworzenia macierzy

     [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    7   10   13
[2,]    2    5    8   11   14
[3,]    3    6    9   12   15

> matrix(x, ncol=5) #liczba kolumn jest wystarczająca do utworzenia macierzy

     [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    7   10   13
[2,]    2    5    8   11   14
[3,]    3    6    9   12   15

> matrix(1:15, ncol=5) #można wpisać bezpośrednio wektor liczb od 1 do 15

     [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    7   10   13
[2,]    2    5    8   11   14
[3,]    3    6    9   12   15
```

Traktowanie przez **R** macierzy jako wektorów (z dodatkowym atrybutem) oznacza, że podobnie jak tam, typów nie można mieszać (zob. str. 7).

Operacje na macierzach

Operacji na macierzach można dokonywać w podobny sposób jak na wektorach. Dodawanie, odejmowanie, mnożenie i dzielenie odbywa się zgodnie z zasadą: element po elemencie.

```
> mac1 <- matrix(c(1, 2, 3, 4, 5, 6), nrow=2) #tworzymy pierwszą macierz
> mac2 <- matrix(c(10, 25, 35, 40, 15, 60), nrow=2) #tworzymy drugą macierz
> mac1

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

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

> mac2

     [,1] [,2] [,3]
[1,]   10   35   15
[2,]   25   40   60

> mac1 + mac2

     [,1] [,2] [,3]
[1,]   11   38   20
[2,]   27   44   66

> mac1 - mac2

     [,1] [,2] [,3]
[1,]   -9  -32  -10
[2,]  -23  -36  -54

> mac2/mac1

     [,1]    [,2] [,3]
[1,] 10.0 11.6667    3
[2,] 12.5 10.0000   10

> mac1*mac2 #to nie jest mnożenie macierzy znane z algebry

     [,1] [,2] [,3]
[1,]   10  105   75
[2,]   50  160  360
```

Wykonując operacje na macierzach dość często wykorzystujemy operator mnożenia macierzy (%*%), funkcję transpozycji (`t()`), funkcję zwracającą elementy diagonalne (`diag()`) oraz funkcję zwracającą macierz odwrotną (`solve()`).

```
> (X <- matrix(round(rnorm(16), 1), nrow=4)) #tworzymy macierz

     [,1] [,2] [,3] [,4]
[1,]  0.1  0.0 -1.9  0.0
[2,]  1.6 -0.6 -0.3  0.0
[3,] -0.5  0.2 -0.9 -0.2
[4,] -0.4 -1.2  0.7 -0.9

> X %*% X #mnożenie

      [,1]  [,2]  [,3] [,4]
[1,]  0.96 -0.38  1.52 0.38
[2,] -0.65  0.30 -2.59 0.06
[3,]  0.80 -0.06  1.56 0.36
[4,] -1.95  1.94 -0.14 0.67

> t(X) #transpozycja: zamiana wierszy z kolumnami

     [,1] [,2] [,3] [,4]
[1,]  0.1  1.6 -0.5 -0.4
[2,]  0.0 -0.6  0.2 -1.2
[3,] -1.9 -0.3 -0.9  0.7
[4,]  0.0  0.0 -0.2 -0.9
```

```
> diag(X) #elementy na przekątnej

[1]  0.1 -0.6 -0.9 -0.9

> solve(X) #macierz odwrotna

          [,1]       [,2]       [,3]       [,4]
[1,] -0.886173  1.0160428  1.3063407 -0.2902979
[2,] -2.076649  1.0160428  3.4491979 -0.7664884
[3,] -0.572956  0.0534759  0.0687548 -0.0152788
[4,]  2.717087 -1.7647059 -5.1260504  0.0280112

> X %*% solve(X) # powinniśmy otrzymać macierz jednostkową (ale numeryczna dokładność)

              [,1]         [,2]        [,3]         [,4]
[1,]  1.00000e+00  0.00000e+00 0.00000e+00  0.00000e+00
[2,]  1.11022e-16  1.00000e+00 2.11636e-16 -3.38271e-17
[3,] -1.11022e-16  0.00000e+00 1.00000e+00 -3.03577e-17
[4,]  4.44089e-16 -2.22045e-16 0.00000e+00  1.00000e+00
```

Do elementów macierzy odwołujemy się, podobnie jak w wypadku wektorów, za pomocą operatora indeksowania [. Podajemy dwa indeksy oddzielone przecinkiem. Pierwszy — odnosi się do numerów wiersza, wiersza, drugi — wskazuje na numery kolumn. Brak indeksu (miejsce puste) oznacza wybór wszystkich wierszy lub/i Poniższe przykłady wyjaśniają istotę wyboru podmacierzy.

```
> set.seed(777) #ziarno generatora
> los <- sample(1:10, size=70, replace=TRUE) #generujemy wektor
> x <- matrix(los, nrow=7, ncol=10) #tworzymy macierz
> x[,] #wybierz wartości ze wszystkich wierszy i wszystkich kolumn; równoważne z: x

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

> x[2, 5] #wybierz wartości z drugiego wiersza i piątej kolumny

[1] 8

> x[, 5] #wybierz  wartości ze wszystkich wierszy i piątej kolumny

[1] 10  8  9  8  9  4  3

> x[1, ] #wybierz wartości z pierwszego wiersza i wszystkich kolumn

 [1]  7  2  9  7 10  4  6  2  1  1

> x[, 5:7] #wybierz wartości ze wszystkich wierszy i kolumn od 5 do 7

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

```
> x[, c(5, 7)] #wybierz wartości ze wszystkich wierszy i kolumn 5 i 7

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

> x[, -5] #wybierz wszystkie wartości, pomijając kolumnę 5

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

> x[c(2, 6), -c(5, 7)] #wiersz 2 i 6 oraz wszystkie kolumny oprócz kolumn 5 i 7

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

Użyteczne funkcje dla macierzy

W tej części omówią najczęściej wykorzystywane funkcje operujące na macierzach.

Funkcja	Opis
`dim(x)`	Wymiar macierzy w postaci wektora: liczba wierszy i kolumn
`ncol(x)`	Liczba kolumn
`nrow(x)`	Liczba wierszy
`cbind(x, y)`	Łączy kolumnowo dwie macierze (lub 2 wektory) w jedną
`rbind(x, y)`	Łączy wierszowo dwie macierze (lub 2 wektory) w jedną
`apply(x, 1 lub 2, fun)`	Wykonuje dla każdego wiersza (gdy 1) lub kolumny (gdy 2) macierzy **X** operację zdefiniowaną przez funkcję fun, np. aby obliczyć sumę każdej kolumy: `apply(x, 2, sum)`

```
> (X <- matrix(1:20, nrow=4, ncol=5))

     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20

> dim(X) #wymiar macierzy, wartość: pierwsza - liczba wierszy, druga - liczba kolumn

[1] 4 5
```

```
> ncol(X)

[1] 5

> nrow(X)

[1] 4

> cbind(X, c(-1, -3, -4, -6)) # połącz kolumnowo macierz X i wektor

     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    5    9   13   17   -1
[2,]    2    6   10   14   18   -3
[3,]    3    7   11   15   19   -4
[4,]    4    8   12   16   20   -6

> (Y <- matrix(-c(1:12), nrow=4)) #

     [,1] [,2] [,3]
[1,]   -1   -5   -9
[2,]   -2   -6  -10
[3,]   -3   -7  -11
[4,]   -4   -8  -12

> cbind(X, Y)

     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    1    5    9   13   17   -1   -5   -9
[2,]    2    6   10   14   18   -2   -6  -10
[3,]    3    7   11   15   19   -3   -7  -11
[4,]    4    8   12   16   20   -4   -8  -12
```

Funkcja `apply()` zasługuje na osobne omówienie. Wyobraźmy sobie sytuację. w której chcemy wykonać identyczne operacje na każdej kolumnie (bądź wierszu). Załóżmy, że będzie to operacja sumowania. Jak mogłyby wyglądać kroki?

```
> ##Chcemy zsumować kolumny w macierzy (nich liczba kolumn = 2)
> ## Następnie wyniki sumy zapisać w obiekcie sumaKolumn
> (x <- matrix(1:8, ncol=2)) #przykładowa macierz

     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8

> kol1 <- sum(x[, 1]) #wybieramy pierwszą kolumnę i sumujemy jej wartości
> kol2 <- sum(x[, 2]) #to samo robimy dla kolumny 2
> sumaKolumn <- c(kol1, kol2) #łączymy dwie sumy i zapisujemy
> sumaKolumn

[1] 10 26
```

A co jeśli kolumn mamy dużo więcej[1]? W takich właśnie sytuacjach możemy wspomagać się funkcją `apply()`, która z kolei wywołuje inną funkcję operującą na każdym wierszu lub kolumnie.

[1]Oczywiście pętla rozwiązuje problem (o tym w rozdz. 3.2). Ale użycie tej funkcji zwiększa czytelność programów

```
> ## Dla wcześniejszego przykładu
> sumaKolumn <- apply(x, 2, sum) #operacje na kolumnach, bo 2
> sumaKolumn

[1] 10 26

> apply(x, 1, sum) #operacje na wierszach bo 1

[1]  6  8 10 12
```

Za funkcję można przyjąć jedną z wbudowanych w **R** (spróbuj dla `var()`, `mean()`, `median()`) lub zdefiniować własną (zob. rozdz. 3.3).

2.3. Czynniki

Zmienne nominalne lub porządkowe, a więc zmienne niemetryczne, traktowane są w **R** (zazwyczaj) jako zmienne typu czynnikowego. To z kolei determinuje sposób w jaki dane będą analizowane czy prezentowane graficznie. W wypadku ramek danych (zob. rozdz. 2.4) domyślnie każda zmienna typu znakowego traktowana jest jako czynnik. Jeśli zdefiniujemy wektor typu znakowego, a chcemy, aby **R** traktował go jako czynnik, wtedy musimy użyć funkcji `factor()` do jej utworzenia.

```
> ## Definiujemy wektor
> (mieszka <- c("miasto", "miasto", "wieś", "miasto", "wieś"))

[1] "miasto" "miasto" "wieś"  "miasto" "wieś"

> typeof(mieszka) #jakiego typu jest wektor

[1] "character"

> str(mieszka) #wyświetl strukturę obiektu

 chr [1:5] "miasto" "miasto" "wieś" "miasto" "wieś"
```

Zobaczmy teraz co się zmieni po wywołaniu funkcji `factor()`.

```
> ## Przekształcamy wektor na czynnik
> mieszka <- factor(mieszka) #przekształć na czynnik, a wynik przypisz zm. mieszka
> mieszka

[1] miasto miasto wieś  miasto wieś
Levels: miasto wieś

> typeof(mieszka) #jaki typ obiektu

[1] "integer"

> str(mieszka) #wyświetl strukturę obiektu

 Factor w/ 2 levels "miasto","wieś": 1 1 2 1 2
```

Co wynika z tej krótkiej analizy? Po pierwsze — zgodnie z zamierzeniem, zmienna `mieszka` jest czynnikiem o dwóch poziomach (*levels*). Po drugie — czynnik reprezentowany jest przez zbiór liczb całkowitych dodatnich (tutaj 1 i 2) z atrybutem `levels` (tutaj miasto i wieś). Po trzecie wreszcie — brane są kolejne liczby, (zaczynając od 1) którym przyporządkowuje się poziomy posortowane alfabetycznie. Dlatego kodowanie wygląda tak: 1 — `miasta`, 2 — `wieś`.

Rozważmy jeszcze sytuację, w której mamy wektor liczbowy edu odnoszący się do trzech poziomów wykształcenia

```
> ## Generujemy wektor o wartościach ze zbioru: 1,2,3
> set.seed(1234)
> (edu <- sample(1:4, 20, replace=TRUE ))

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

Załóżmy, że kodowanie przebiega według schematu: 1 — podstawowe, 2 — średnie, 3 — licencjat, 4 — magisterium. Celem stworzenie czynnika o poziomach opisujących wykształcenie. Zauważmy, że postępując podobnie jak w wypadku zmiennej zmiennej `mieszka`, a więc wywołując funkcję `factor(edu)` zyskujemy niewiele — dalej nie powiedzieliśmy **R** o tych poziomów (podstawowe, średnie itd.). Możemy to zrobić wywołując wspomnianą funkcję z dodatkowymi argumentami: `levels` oraz `labels` (poziomy oraz etykiety). Dodatkowo uwzględnienie argumentu `order` z wartością `TRUE` poinformuje **R**, że mamy do czynienia ze zmienną porządkową.

```
> #zmienną edu przekształcamy na czynnik
> eduOrd <- factor(edu, levels=c(1:4),
+                  labels= c("podstawowe", "średnie",  "licencjat",  "magisterium"),
+                  order=TRUE)
> eduOrd

 [1] podstawowe  licencjat   licencjat   licencjat   magisterium licencjat   podstawowe
 [8] podstawowe  licencjat   licencjat   licencjat   licencjat   średnie     magisterium
[15] średnie     magisterium średnie     średnie     podstawowe  podstawowe
Levels: podstawowe < średnie < licencjat < magisterium
```

W praktyce spotykamy się z potrzebą usunięcia jednego lub kilku poziomów zmiennej. W kontekście powyższego przykładu, nich będzie to poziom: `podstawowe`. Usunięcie tych elementów wektora `eduOrd`, którym odpowiada ten poziom, nie powoduje automatycznej korekty atrybutu `levels` — dalej będą 4 poziomy. W dalszych analizach ten niechciany poziom będzie występował, co pokazuje poniższy przykład:

```
> (eduOrd2 <- eduOrd[eduOrd != "podstawowe"]) #usuń wartości: podstawowe

 [1] licencjat   licencjat   licencjat   magisterium licencjat   licencjat   licencjat
 [8] licencjat   licencjat   średnie     magisterium średnie     magisterium średnie
[15] średnie
Levels: podstawowe < średnie < licencjat < magisterium

> table(eduOrd2) #tabela liczebności zawiera podstawowe

eduOrd2
 podstawowe     średnie   licencjat magisterium
          0           4           8           3
```

Aby usunąć nieużywane poziomy, należy posłużyć się funkcją `droplevels()`.

```
> (eduOrd2 <- droplevels(eduOrd2)) #usuń nieużywane poziomy

 [1] licencjat   licencjat   licencjat   magisterium licencjat   licencjat   licencjat
 [8] licencjat   licencjat   średnie     magisterium średnie     magisterium średnie
[15] średnie
Levels: średnie < licencjat < magisterium

> table(eduOrd2) #tabela liczebności

eduOrd2
   średnie   licencjat magisterium
         4           8           3
```

2.4. Ramki danych

Ramki danych, identycznie jak macierze, mają strukturę dwuwymiarową, na którą składają się wiersze i kolumny. Istotnym elementem odróżniającym je od macierzy jest możliwość mieszania typów. Tym samym, przykładowo, jedna kolumna może składać się z liczb rzeczywistych odnoszących się do liczby ludności, a druga zawierać nazwy miejscowości. Tak zorganizowany zbiór danych znany jest np. z arkuszy kalkulacyjnych czy innych programów statystycznych (np. IBM SPSS). Wtedy wiersze traktujemy jak przypadki, obserwacje itp. Kolumny z kolei odnoszą się do zmiennych.

W **R** obiekt o takiej strukturze tworzymy używając funkcji `data.frame(col1, col2, ...)`, w której każdy `col` jest wektorem kolumnowym o dowolnym typie oraz identycznej długości. Jeśli wektory będą różnić się długością, **R** zastosuje regułę zawijania.

```
> ## Definiujemy wektory i tworzymy ramkę danych
> sok <- c("kubus", "pysio", "leon", "bobo frut")
> cena <- c(1.2, 1.35, 1.65, 1.99)
> cukier <- c(11.5, 12, 10, 9.6)
> dfSok <- data.frame(cena, cukier, sok)
> dfSok

  cena cukier       sok
1 1.20   11.5     kubus
2 1.35   12.0     pysio
3 1.65   10.0      leon
4 1.99    9.6 bobo frut

> ## Drugi, możliwy sposób
> (dfSok2 <- data.frame(c(1.2, 1.35, 1.65, 1.99), c(11.5, 12, 10, 9.6),
+ c("kubus", "pysio", "leon", "bobo frut")) )

  c.1.2..1.35..1.65..1.99. c.11.5..12..10..9.6.
1                     1.20                11.5
2                     1.35                12.0
3                     1.65                10.0
4                     1.99                 9.6
  c..kubus....pysio....leon....bobo.frut..
1                                   kubus
2                                   pysio
3                                    leon
4                                bobo frut
```

Każda kolumna ma swoją nazwę. Jeśli tych nazw nie podamy, wtedy **R** domyślnie przyjmie jakieś nazwy, tak jak to zrobił w wypadku zmiennej `dfSok2`. Na szczęście nazwy zmiennych można zmienić przy użyciu funkcji `names()`:

```
> names(dfSok) #podaj nazwy zmiennych dfSok

[1] "cena"   "cukier" "sok"

> names(dfSok2) #paskudne nazwy, które trzeba zmienić

[1] "c.1.2..1.35..1.65..1.99."
[2] "c.11.5..12..10..9.6."
[3] "c..kubus....pysio....leon....bobo.frut.."

> names(dfSok2) <- c("cena", "ile_cukru", "soczek") #za nazwy podstaw wektor
> dfSok2 #ramka danych ze zmienionymi nazwami
```

```
  cena ile_cukru    soczek
1 1.20      11.5     kubus
2 1.35      12.0     pysio
3 1.65      10.0      leon
4 1.99       9.6 bobo frut
```

Do elementów ramki danych można odwołać się w taki sam sposób, jak do elementów macierz (zob. str. 17). Istnieją jednak alternatywne sposoby, z którymi warto się zapoznać:

- `moja_ramka$nazwa_kolumny` — po nazwie ramki umieszczamy operator `$`, po którym z kolei podajemy nazwę kolumny;
- `moja_ramka[numer_kolumny]` — ramka danych jest przypadkiem szczególnym listy (zob. rozdz. 2.5), dlatego taki sposób odwołania jest właściwy (dla macierzy nie). Zwróćmy uwagę na wynik tej operacji — pojedynczy `[` zwraca zawsze obiekt tej samej klasy (tutaj *data frame*). Jeśli użyjemy podwójnego operatora `[[` otrzymamy wektor. (uwaga: można też `moja_ramka[ ,numer_kolumny]`)
- `moja_ramka["nazwa_kolumny"]` — odwołanie następuje poprzez nazwę kolumny/zmiennej ujętej w cudzysłowy. Zamiast nazwy jednej kolumny można podać wektor nazw. (uwaga: można też `moja_ramka[ ,"nazwa_kolumny"]`)

```
> ## Wykorzystamy poprzednią ramkę danych: dfSok
> dfSok[, 1] #już znany sposób

[1] 1.20 1.35 1.65 1.99

> dfSok$cena # weź kolumnę z ceną

[1] 1.20 1.35 1.65 1.99

> dfSok[1] #wez pierwsza kolumnę, czyli cenę; wynikiem jest data frame

  cena
1 1.20
2 1.35
3 1.65
4 1.99

> dfSok[[1]] #jak wyżej, ale wynikiem jest wektor

[1] 1.20 1.35 1.65 1.99

> dfSok["cena"] #identyczne z dfSok[ ,"cena"]

  cena
1 1.20
2 1.35
3 1.65
4 1.99

> dfSok[c(1,3)]

  cena       sok
1 1.20     kubus
2 1.35     pysio
3 1.65      leon
4 1.99 bobo frut

> dfSok[c("cukier", "sok")]

  cukier       sok
1   11.5     kubus
2   12.0     pysio
3   10.0      leon
4    9.6 bobo frut
```

Do ramek danych można dodawać kolumny i wiersze w taki sam sposób jak w wypadku macierzy, a więc używając funkcji `cbind()` i `rbind()`. Kolumny można też dodawać —- wykorzystując operator `$` lub usuwać — przypisując kolumnie wartość `NULL`. Jeśli chcemy dodać kolumnę o nazwie `nowaKol` do ramki `mojeDane` o wartościach [1,5,2,4,3] oraz usunąć kolumnę `staraKol`, należy napisać:

```
mojeDane$nowaKol <- c(1, 5, 2, 4, 3)
mojeDane$staraKol <- NULL
```

W RStudio możemy korzystać z podpowiedzi. Po wpisaniu nazwy zmiennej, a następnie operatora `$` naciśnijmy klawisz *Tab*. Powinna pojawić się lista, z której możmy wybrac interesującą nas zmienną. Przy tej okazji warto wspomnieć, że jeśli zaczniemy pisać nazwę funkcji i wciśniemy *Tab*, wtedy pojawi się lista wszystkich dostępnych funkcji, rozpoczynających się od wpisanego ciągu. Z kolei po napisaniu nazwy funkcji i naciśnięciu *Tab* pojawi się jej lista argumentów. Proszę wpisać `rn` i nacisnąć *Tab*, a później spróbować dla `rnorm(`.

2.5. Listy

Listę mogą tworzyć elementy dowolnego typu. Z tego względu uważamy ją za najbardziej złożoną strukturę w **R**. Bardzo często jest wykorzystywana do przechowywania różnego typu danych czy informacji. Przykładowo, weryfikując hipotezę statystyczną testem t-Studenta musimy wywołać odpowiednią funkcję (`t.test()`). Okazuje się, że jej wynikiem jest lista składająca się z 9 elementów, którymi przykładowo są: wartość statystyki, przedział ufności, postać hipotezy alternatywnej, p-wartość. Nie jest możliwe zapisanie tych informacji w obiektach omówionych wcześniej, np. w ramce danych, bo mamy obiekty różnej długości: przedział ufności zawiera dwie wartości, natomiast wartość statystyki jedną. W takich właśnie sytuacjach potrzebujemy listy.

Aby utworzyć listę, należy użyć funkcji `list()`. Ogólnie można przyjąć, że `list(obiekt1, obiekt2, ...)` utworzy listę, w której każdy `obiekt` może być wektorem, macierzą, ramką danych czy nawet listą. Zauważmy, że jeśli każdy obiekt jest wektorem takiej samej długości, to mamy strukturę odpowiadającą ramce danych — być może w takiej sytuacji wcale nie potrzebujemy listy.

```
> ## Przykład - tworzenie listy
> mojaLista <- list("To moja pierwsza lista",
+                   c(20, 10, 15, 16),
+                   c("Ewa", "Nella", "Tammy"),
+                   matrix(1:10, nrow=2))
> mojaLista

[[1]]
[1] "To moja pierwsza lista"

[[2]]
[1] 20 10 15 16

[[3]]
[1] "Ewa"   "Nella" "Tammy"

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

Powyższa lista składa się z 4 obiektów różnego typu (jakich?). Chcąc odwołać się do elementów listy wykorzystujemy znane operatory: `[` lub `[[`. Pierwszy z nich, jak wspomniano wcześniej, zwraca element tego samego typu co obiekt główny (tutaj lista). W wyniku użycia drugiego do naszego przykładu otrzymamy odpowiednio: wektor, wektor, wektor, macierz.

```
> #Wybór elementów z listy
> mojaLista[2] #otrzymamy listę; pojawi się charakterystyczny podwójny nawias: [[1]]

[[1]]
[1] 20 10 15 16

> mojaLista[[2]] #otrzymamy wektor

[1] 20 10 15 16

> mojaLista[c(2,3)] # uwaga: mojaLista[[c(2,3)]] jest równe mojaLista[[2]][3]

[[1]]
[1] 20 10 15 16

[[2]]
[1] "Ewa"   "Nella" "Tammy"
```

Do elementów listy można odwoływać się wykorzystując nazwy. Odbywa się to w identyczny sposób jak w wypadku ramek danych, a więc z wykorzystaniem operatora $. Nazwijmy więc poszczególne elementy listy:

```
> names(mojaLista) <- c("tytul", "cena", "imie", "mojamac")
> mojaLista #co się zmieniło

$tytul
[1] "To moja pierwsza lista"

$cena
[1] 20 10 15 16

$imie
[1] "Ewa"   "Nella" "Tammy"

$mojamac
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

```
> mojaLista$imie #wybór jednego elementu

[1] "Ewa"   "Nella" "Tammy"

> mojaLista[c("imie", "cena")] #wybór dwóch elementów listy

$imie
[1] "Ewa"   "Nella" "Tammy"

$cena
[1] 20 10 15 16
```

Działania na listach — funkcja `lapply` i `sapply`

Nierzadko chcemy wykonać identyczne operacje na wszystkich elementach listy. Idea jest identyczna jak w wypadku funkcji `apply()`, opisanej na str. 18, z ta różnicą że operacji nie wykonujemy na wierszach bądź kolumnach. W takiej sytuacji możemy wykorzystać funkcję `lapply()`, która zwraca obiekt będący

zawsze listą. Jeśli wiemy, że wynikiem może być jakaś prostsza struktura, np. macierz, wektor, wtedy lepiej użyć funkcji `sapply()`. Pierwsze litera `s` w tej funkcji oznacza uproszczenie (*simplified*). W obu funkcjach pierwszym argumentem jest obiekt typu lista, drugim funkcja. Zobaczmy jak działają.

```
> #Weźmiemy poprzednią listę, wyłączając ostatni element
> (mylis <- mojaLista[1:3]) #tylko elementy od 1 do 3

$tytul
[1] "To moja pierwsza lista"

$cena
[1] 20 10 15 16

$imie
[1] "Ewa"   "Nella" "Tammy"

> lapply(mylis, length) #oblicz dlugosc (length) kazdago elementu listy

$tytul
[1] 1

$cena
[1] 4

$imie
[1] 3

> sapply(mylis, length) #jak wyżej, ale uprości do wektora

tytul  cena  imie
    1     4     3

> sapply(mylis, class) #jakiej klasy są elementy listy

      tytul        cena        imie
"character"   "numeric" "character"
```

Musimy pamiętać o tym, że funkcja wykonuje takie same operacje na każdym elemencie listy, więc dla każdego elementu muszą mieć one sens. W powyższym przykładzie nie ma sensu użycie funkcji `mean` obliczającej średnią.

2.6. Funkcje R w rachunku prawdopodobieństwa

Podstawowy pakiet `stats`, który wczytywany jest zawsze podczas uruchamiania **R**, zawiera wiele funkcji rozkładów prawdopodobieństwa. Sięgając do pomocy (wpisz: `?Distributions`), można zapoznać się z tą listą. My wykorzystamy tylko niektóre.

W ramach każdego rozkładu można wyróżnić 4 funkcje: gęstość lub rozkład prawdopodobieństwa (`d` — *density*), dystrybuanta (`p` — *probability*), kwantyle (`q` — *quantile*), generator liczb pseudolosowych (`r` — *random*). Podany w nawiasie przedrostek (wzięty z angielskich nazw) określa, co zwraca funkcja. Przykładowo dla rozkładu normalnego trzonem jest wyraz `norm`, a poprzedzając go przedrostkiem, otrzymamy funkcje: `dnorm()`, `pnorm()`, `qnorm()`, `rnorm()`. W ostatniej przedrostkiem jest `r`, więc funkcja generuje liczby z rozkładu normalnego.

Zobaczmy jak wyglądają funkcje i ich argumenty (niektóre pominąłem) dla wybranych rozkładów.

Rozkład normalny: `mean` — średnia, `sd` — odchylenie standardowe; są wartości domyślne

```
dnorm(x, mean = 0, sd = 1)
pnorm(x, mean = 0, sd = 1)
qnorm(p, mean = 0, sd = 1)
rnorm(n, mean = 0, sd = 1)
```

Rozkład normalny ma dwa parametry: średnią (*mean*) i odchylenie standardowe (*sd*). Gdy tych parametrów nie zmienimy, wtedy przyjmowane są wartości domyślne, widoczne powyżej w definicji funkcji. Argumentami na pierwszej pozycji są: `x` — punkt w którym chcemy obliczyć wartość funkcji gęstości i wartość dystrybuanty, `p` — prawdopodobieństwo (rząd kwantyla), `n` — ile wygenerować obserwacji. Działanie pierwszych trzech funkcji zilustruję przykładem. Generowanie liczb losowych pozostawiam jako ćwiczenie.

```
> ## Przykład dla funkcji rozkładu normalnego
> ## Uwaga1: jeśli zachowamy kolejność wpisywania, słowa mean i sd można pominąć
> ## Uwaga2: zawsze w zapisie N(a, b), b jest wariancją, dlatego odch.stand. do kwadratu
> dnorm(0) # średnia i odch. standardowe domyślne, czyli 0 i 1

[1] 0.398942

> dnorm(0, mean=10, sd=15)

[1] 0.0212965

> pnorm(0) #Pr(X <= 0), gdzie X - N(0, 1)

[1] 0.5

> pnorm(3, 6, 10) #Pr(X <= 3), gdzie X - N(6, 10^2)

[1] 0.382089

> qnorm(0.7) # oblicz a, by F(a) = 0.7

[1] 0.524401

> qnorm(0.7, 50, 25) # oblicz a, by F(a) = 0.7 ale X - N(50, 25^2)

[1] 63.11
```

Za pierwszy argument podstawialiśmy jedną wartość. Tym argumentem, co nie powinno dziwić, może być wektor. Wtedy funkcja wykonuje operacje na każdym elemencie wektora, a liczba zwracanych wartości jest równa długości wektora.

```
> pnorm(c(-1, -0.5, 2.1, 3.5), mean=0.5, sd=3) # Oblicza prawdop. dla każdego elementu

[1] 0.308538 0.369441 0.703099 0.841345
```

Rozkład t-Studenta: `df` — liczba stopni swobody; nie ma domyślnej

```
dt(x, df)
pt(x, df)
qt(p, df)
rt(n, df)
```

Rozkład jednostajny: `min` i `max` — odpowiednio dolny i górny kraniec przedziału; są wartości domyślne

```
dunif(x, min = 0, max = 1)
punif(x, min = 0, max = 1)
qunif(p, min = 0, max = 1)
runif(n, min = 0, max = 1)
```

Rozkład wykładniczy: `rate` — parametr rozkładu (lambda); są wartości domyślne

```
dexp(x, rate = 1)
pexp(x, rate = 1)
qexp(p, rate = 1)
rexp(n, rate = 1)
```

Rozkład normalny, t-studenta, jednostajny i wykładniczy należą do rodziny rozkładów ciągłych. Przedstawiony opis funkcji, a dotyczący przedrostków, ma również zastosowanie do rozkładów dyskretnych. Pamiętać jednak należy o jednej istotnej różnicy. Otóż wszystkie funkcje z przedrostkiem `d` (*density*), dla rozkładów dyskretnych, zwracają wartość prawdopodobieństwa. Powtórzmy: dla ciągłych — wartość funkcji gęstości, dla dyskretnych — wartość prawdopodobieństwa.

Rozkład dwumianowy: `size` — liczba prób (eksperymentów), `prob` — prawdopodobieństwo sukcesu w pojedynczej próbie; nie ma wartości domyślnej

```
dbinom(x, size, prob)
pbinom(x, size, prob)
qbinom(p, size, prob)
rbinom(n, size, prob)
```

Obliczmy prawdopodobieństwo tego, że w grupie 20 studentów przynajmniej 7 wyjedzie na wakacje za granicę. Przyjmujmy, że prawdopodobieństwo tego zdarzenia dla losowo wybranego studenta wynosi 0.3.

```
> dbinom(7, size=20, prob=0.3) #Pr(X=7) a my chcemy Pr(X>=7)

[1] 0.164262

> (prawd <- dbinom(7:20, size=20, prob=0.3)) #Pr(X=7), Pr(X=8), ..., Pr(X=20)

 [1] 1.64262e-01 1.14397e-01 6.53696e-02 3.08171e-02 1.20067e-02 3.85928e-03 1.01783e-03
 [8] 2.18107e-04 3.73898e-05 5.00756e-06 5.04964e-07 3.60688e-08 1.62717e-09 3.48678e-11

> sum(prawd) # Sumujemy i to jest nasza odpowiedź

[1] 0.39199
```

Można to zadanie rozwiązać, wykorzystując dystrybuantę: $F(x) = \Pr(X \leqslant x)$. W zadaniu mamy obliczyć:

$$\Pr(X \geqslant 7) = \Pr(X > 6) = 1 - \Pr(X \leqslant 6) = 1 - F(6)$$

więc

```
> 1 - pbinom(6, 20, 0.3)

[1] 0.39199
```

Rozkład Poissona: `lambda` — parametr w rozkładzie poissona; nie ma wartości domyślnej

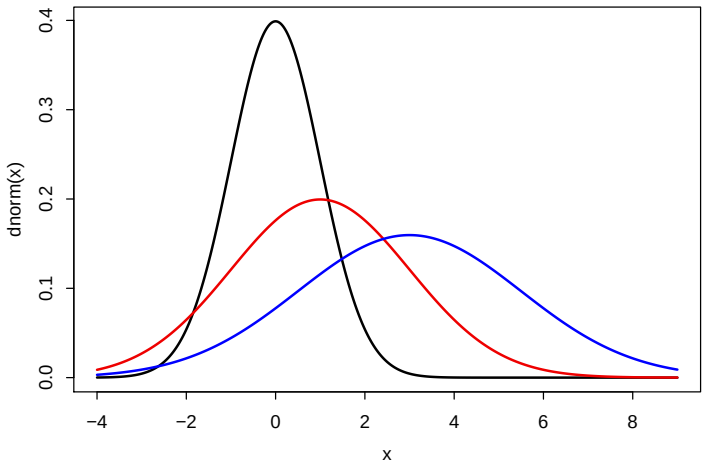
```
dpois(x, lambda)
ppois(x, lambda)
qpois(p, lambda)
rpois(n, lambda)
```

Na zakończenie tego rozdziału podam sposób rysowania funkcji gęstości i rozkładu prawdopodobieństwa. Bardziej szczegółowe omówienie zagadnienia wizualizacji danych zamieszczam w rozdz. 5.2.

Aby narysować wykres musimy mieć zbiór punktów (x, y). Wiemy, że zmienna losowa o rozkładzie normalnym przyjmuje wartości ze zbioru wszystkich liczb rzeczywistych, czyli $x \in (-\infty, \infty)$. Gdy celem jest wykres, zbiór ten ograniczamy do wąskiego zakresu, np. dla rozkładu zestandaryzowanego wystarczy przyjąć przedział $(-4, 4)$. Z kolei wartości funkcji gęstości w punktach x obliczymy wykorzystując znaną już funkcję `dnorm()`.

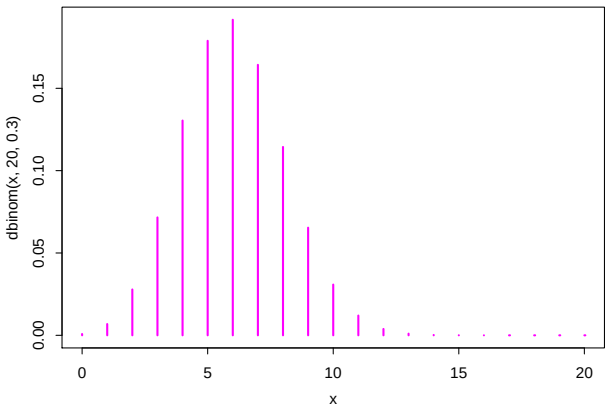
Do narysowania wykresu użyjemy funkcji `plot()`. Pierwszy argument to x, drugi to y. Gdy na tym samym rysunku chcemy nanieść inne funkcje, nie możemy ponownie użyć funkcji `plot()` (spróbuj). Dlatego kolejne gęstości naniesiemy używając funkcji `lines()`. Parametry graficzne, które zamieściłem w pierwszej linijce służą poprawie efektu końcowego (jest to nieistotne teraz). Efekt poniżej.

```
> # Jak narysować wykresy kilku funkcji gęstości
> par(mar = c(4, 4, .1, .1), cex.lab = .95, cex.axis = .9, mgp = c(2, .7, 0), tcl = -.3)
> x <- seq(-4, 9, by=0.01) #przedział: jakie wartości przyjmuje x
> plot(x, dnorm(x), type="l", lwd=2) # "l" - linia, 2 - grubość lini
> lines(x, dnorm(x, 1, 2), lwd=2, col="red2") # "red2" - kolor
> lines(x, dnorm(x, 3, 2.5), lwd=2, col="blue") #"blue" - kolor
```



Jeśli celem jest narysowanie rozkładu prawdopodobieństwa zmiennej losowej, postępujemy bardzo podobnie. Też potrzebujemy wartości x jakie przyjmuje zmienna losowa oraz prawdopodobieństwa $\Pr(X = x)$. Narysujmy rozkład prawdopodobieństwa dla przykładu ze str. 28. Wiemy, że $x \in \{0, 1, 2, \ldots, 20\}$ natomiast prawdopodobieństwa policzymy przy użyciu funkcji `dbinom()`.

```
> par(mar = c(3, 3, .1, .1), cex.lab = .95, cex.axis = .9, mgp = c(2, .7, 0), tcl = -.3)
> x <- 0:20 #wartości jakie przyjmuje x
> plot(x, dbinom(x, 20, 0.3), type="h", lwd=2, col="magenta") # "h" - linia pionowa
```



2.7. Zadania

Obsługa R

Zad. 1. Wykorzystując odpowiednie funkcje, stwórz wektory, które będą miały następujące elementy:

a) $1, 4, 6, 13, -10$
b) $1, 3, 5, \ldots, 101$
c) $4, 4, 4, 4, 7, 7, 7, 7, 9, 9, 9, 9$
d) $4, 7, 9, 4, 7, 9, 4, 7, 9, 4, 7, 9, 4, 7, 9, 4, 7, 9.$

Następnie dla każdego podaj: długość (liczba elementów), element najmniejszy i największy. Oczywiście należy skorzystać z odpowiednich funkcji, a nie liczyć.

Zad. 2. Wykorzystaj poniższy skrypt do wygenerowania wektora cena (przyjmujemy, że jest w PLN).

```
set.seed(1313)
cena <- rnorm(100, mean=50, sd=10)
```

Następnie zaokrąglij cenę do 2 miejsc po przecinku. Zdefiniuj nowy wektor, którego wartości będą ceną wyrażoną w EURO; przyjmij kurs wymiany na poziomie 4.19 PLN/EUR. Nowy wektor zaokrąglij do liczb całkowitych, a następnie:

a) znajdź wartość największą i najmniejszą;
b) podaj liczbę elementów unikalnych, posortuj te wartości i wyświetl w konsoli **R**;
c) wykorzystując wzory oblicz: sumę elementów ($\sum_{i=1}^{n} x_i$), średnią arytmetyczną ($\frac{1}{n}\sum_{i=1}^{n} x_i$) i geometryczną ($\sqrt[n]{x_1 \cdot x_2 \cdot \ldots x_n}$).
d) podaj liczbę wartości: (i) większą od 13 EUR, (ii) mniejszą od 10 EUR i większą od 15 EUR.
e) utwórz ramkę danych składającą się z ceny w PLN i EUR. Za nazwy kolumn przyjmij odpowiednio: `cenaPLN`, `cenaEUR`.

Zad. 3. Zaciągnięto kredyt hipoteczny w wysokości K PLN na okres L lat. Spłata następuje w cyklu miesięcznym, w równych ratach przy rocznej stopie oprocentowania równej r. Szczegóły zawiera poniższy skrypt, który należy skopiować do swojego pliku.

```
r <- 0.05 #oprocentowanie roczne
rr <- 1+r/12
K <- 300000 #kwota kredytu
L <- 20 #ile lat
N <- 12*L #liczba rat (ile miesięcy)
n <- 1:N #wektor zawierający kolejne okresy
rataKredytu <- K*rr^N*(rr-1)/(rr^N-1)
zadluzenie <- K*(rr^N-rr^n)/(rr^N-1)
odsetki <- K*(rr^N-rr^(n-1))/(rr^N-1)*(rr-1)
rataKapitalu <- rataKredytu - odsetki
kredyt <- cbind(rataKapitalu, odsetki, rataKredytu, zadluzenie) #
```

Przeanalizuj działanie powyższego programu zwracając uwagę na regułę zawijania: chodzi o sposób wykonywania operacji przez **R**, a nie analizę istoty i sensu wzorów. Utwórz macierz o nazwie `kredyt`, której kolumnami będą następujące wektory: `rataKapitalu, odsetki, rataKredytu, zadluzenie`. Użyj funkcji `class()` by sprawdzić, czy utworzony obiekt faktycznie jest macierzą. Następnie wykorzystując odpowiednie funkcje:

a) użyj funkcji do wyświetlenia pierwszych i ostatnich 10 wierszy: `head()` i `tail()`.
b) jaki jest wymiar macierzy.
c) wyświetl w konsoli wiersze dla rat: (i) od 100 do 125 (ii) pierwszych 20 (iii) ostatnich 30 (iv) od 20 do 30 i od 50 do 60 (v) co dziesiątą ratę (10, 20, 30 itd.)
d) oblicz sumaryczną wielkość zapłaconych odsetek, rat kredytu i rat kapitałowych. Jakieś wnioski?
e) pozmieniaj parametry kredytu, np. liczba rat, wysokość kredytu, stopa oprocentowania, i zobacz jak to wpłynie na sumaryczna spłatę (punkt wcześniejszy).
f) (⋆) od którego okresu wysokość raty kapitałowej (`rataKapitalu`) zaczyna przewyższać wysokość spła-

canych odsetek (`odsetki`)? Może przyda się funkcja `which()`.

Zad. 4. Poniższa ramka danych zawiera informacje o masie [kg] i wysokości [cm] ciała.

```
medic <- data.frame(
c(82.5, 65.1, 90.5, 80.9, 74, 74.4, 73.5, 75.6, 70.1, 61.8, 80.6, 82.2, 54.1, 60),
c(181, 169, 178, 189, 178, 175, 173, 187, 175, 165, 185, 178, 162, 185))
```

Zmień domyślne nazwy kolumn na: `masa` i `wysokosc`. Utwórz dodatkową kolumnę (o nazwie `BMI`), której wartościami będzie wskaźnik masy ciała. Wskaźnik ten obliczamy ze wzoru: $\frac{\text{masa[kg]}}{(\text{wysokosc[m]})^2}$. Uwaga: w danych mamy wysokość w centymetrach, a we wskaźniku w metrach. Zobacz dalej efekt końcowy (wyświetlam tylko 4 wiersze)

a) (⋆) utwórz wektor, który będzie przyjmował wartość 1 — gdy BMI < 18.5, wartość 2 — gdy BMI $\in [18.5, 24.99]$, wartość 3 — gdy BMI >24.99. Nazwij ten wektor `waga`.

b) Wektor `waga` z poprzedniego punktu ma postać: 3, 2, 3, 2, 2, 2, 2, 2, 2, 2, 2, 3, 2, 1. Wykorzystując czynniki dokonaj kodowania według schematu: 1 — niedowaga, 2 — prawidlowa, 3 — nadwaga. Pamiętaj, że to zmienna porządkowa. Następnie włącz tą zmienną do ramki danych `medic`. Efekt poniżej (wyświatlam tylko 4 wiersze)

```
> head(medic, 4)
  masa wysokosc      BMI       waga
1 82.5      181 25.1824    nadwaga
2 65.1      169 22.7933 prawidlowa
3 90.5      178 28.5633    nadwaga
4 80.9      189 22.6477 prawidlowa
```

c) Wykorzystując operatory: `[, [[, $` wybierz (według uznania i fantazji) kolumnę/kolumny.

Prawdopodobieństwo

Zad. 5. Załóżmy, że 30% właścicieli komputerów używa systemu Mac, 50% Windowsa a 20% Linuksa. Zaobserwowano, że 65% użytkowników Maca ulega zarażeniu wirusem. W wypadku Windowsa i Linuksa procent infekcji wynosi odpowiednio 82% i 50%. Wybieramy losowo jedną osobę, która okazuje się mieć zainfekowany komputer. Jakie jest prawdopodobieństwo, że jest to użytkownik Windowsa.

Zad. 6. Weźmy pod uwagę rzut symetryczną kostką, gdzie $\Omega = \{1,2,3,4,5,6\}$. Zdefiniujmy zdarzenia $A = \{2,4,6\}$ i $B = \{1,2,3,4\}$. Obliczyć prawdopodobieństwa: $\Pr(A)$, $\Pr(B)$, $\Pr(AB)$. Pokazać analitycznie, że zdarzenia A i B są niezależne. Przeprowadzić również dowód empiryczny, wykonując odpowiedni eksperyment komputerowy, polegający na pobieraniu losowej próby (z przestrzeni zdarzeń) i obliczeniu: $\widehat{\Pr}(A)$, $\widehat{\Pr}(B)$, $\widehat{\Pr}(AB)$. Wykorzystać do tego funcję `sample()`. Porównać wyniki dla różnych wartości próby n ($n = 10, 100, 1000$). Skonfrontować wyniki z wartościami prawdopodobieństwa obliczonego analitycznie.

Zad. 7. Niech X i Y będą zmiennymi losowymi o rozkładzie normalny $\mathcal{N}(0,1)$ i $\mathcal{N}(0,4)$ odpowiednio (w rozkładzie podano σ^2). Narysować obie funkcje gęstości. Wykorzystując interpretację geometryczną prawdopodobieństwa oraz wspomagając się wykresami — nie wykonując żadnych obliczeń — podać lub zdefiniować relację ($>, <, =, \leqslant, \geqslant$)

a) $\Pr(X > 0) = \ldots$

b) $\Pr(Y \geqslant 0) = \ldots$

c) $\Pr(X \geqslant 2) \ldots \Pr(X < -2)$

d) $\Pr(0 < Y \leqslant 3) \ldots \Pr(2 < Y \leqslant 5)$

e) $\Pr(-1 < X \leqslant 1) \ldots \Pr(0 < X \leqslant 2)$

f) $\Pr(0 < Y \leqslant 2) \ldots \Pr(0 < X \leqslant 2)$

g) $\Pr(\frac{3\ln(2)}{8} < Y < 1.5) \ldots \Pr(\frac{3\ln(2)}{8} < X < 1.5)$ Następnie wykorzystując odpowiednie funkcje **R** obliczyć powyższe prawdopodobieństwa.

Zad. 8. Dla zestandaryzowanego rozkładu normalnego i rozkładu t-studenta o 15 stopniach swobody

a) wyznaczyć kwantyle rzędu $p = 0.85$, $p = 0.99$, $p = 0.27$.
b) obliczyć prawdopodobieństwa: $\Pr(X > 1.8)$, $\Pr(X \geqslant 2.47)$.
Do obliczeń wykorzystać załączone tablice, a następnie obliczenia powtórzyć wybierając odpowiednie funkcje **R**.

Zad. 9. Pewien bank ma atrakcyjny program kart kredytowych. Klienci, którzy spełniają wymagania, mogą otrzymać taką kartę na preferencyjnych warunkach. Analiza danych historycznych pokazała, że 35% wszystkich wniosków zostaje odrzuconych ze względu na niespełnienie wymagań. Załóżmy, że przyjęcie lub odrzucenie wniosku jest zmienna losową o rozkładzie Bernoulliego. Jeśli próbę losową stanowi 20 wniosków:
a) narysować rozkład prawdopodobieństwa zmiennej losowej;
b) jakie jest prawdopodobieństwo tego, że dokładnie trzy wnioski zostaną odrzucone;
c) jakie jest prawdopodobieństwo tego, że 10 wniosków zostanie przyjętych;
d) jakie jest prawdopodobieństwo tego, że przynajmniej 10 wniosków zostanie przyjętych.
Wskazówka: rozkład dwumianowy $\Pr(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$, $k = 0, 1, \ldots, n$
Maksymalnie wpomagać się **R**.

Zad. 10. Salon samochodowy rejestruje dzienną sprzedaż nowego modelu samochodu *Shinari*. Wyniki obserwacji doprowadziły do wniosku, że rozkład liczby sprzedanych samochodów w ciągu dnia można przybliżyć rozkładem Poissona:

$$\Pr(X = x|\lambda) = \frac{\lambda^x e^{-\lambda}}{x!}, \qquad x = 0, 1, 2, \ldots$$

z parametrem $\lambda = 5$. Obliczyć (używając **R**) prawdopodobieństwo tego, że salon:
a) nie sprzeda ani jednej sztuki;
b) sprzeda dokładnie 5 sztuk;
c) sprzeda przynajmniej jedną sztukę;
d) sprzeda przynajmniej 2 sztuki ale mniej niż 5;
e) sprzeda 5 sztuk przy założeniu, że sprzedał już ponad 3 sztuki.

Rozdział 3

Wyrażenia warunkowe, pętle, funkcje

W tym rozdziale ograniczam się do absolutnego, choć koniecznego, minimum. Opanowanie tych podstaw, niewspółmiernie do włożonego wysiłku, ułatwi nam pracę i zaoszczędzi czas. Naturalną konsekwencją jest bardziej elastyczny, przejrzysty i czytelny kod.

3.1. Wyrażenia warunkowe: `if...else`, `ifelse`

Wyrażenia warunkowe sterują przepływem wykonywania programu. Tym samym pozwalają zmieniać — w zależności od tego, czy warunek logiczny jest spełniony — kolejności, w jakiej program jest wykonywany. Składnia z użyciem `if` wygląda następująco:

```
if (warunek)
    wykonaj_jesli_TRUE
```

W pierwszym kroku **R** sprawdza jaką wartość logiczną (`TRUE` czy `FALSE`) ma `warunek`. Jeśli jest on prawdziwy, wykonywana jest następna linijka. Oczywiście jest ona pomijana, gdy wyrażenie jest fałszywe, co ilustruje poniższy przykład.

```
> if (2==3)  #Jeżeli 2 jest równe 3 to
+     a <- 10  #za a podstaw 10
> a              #pokaż a; ale takiego obiektu nie ma, więc błąd

Error: object 'a' not found
```

Jeśli chcemy, aby instrukcji do wykonania było więcej, wtedy należy ująć je w nawiasy klamrowe. Wszystkie wyrażenia między tymi nawiasami zostaną poddane ewaluacji. Zobaczmy jak wygląda skłania oraz jej wariant z `else` zamieszczony po prawej stronie:

```
if (warunek) {
    wykonaj1_jesli_TRUE
    wykonaj2_jesli_TRUE
    ........
}
```

```
if (warunek) {
    wykonaj1_jesli_TRUE
    wykonaj2_jesli_TRUE
    ........
} else {
  wykonaj1_jesli_FALSE
  wykonaj2_jesli_FALSE
  ........
  }
```

Jeśli warunek jest spełniony, wykonywane są instrukcje w pierwszym nawiasie klamrowym. W wariancie po prawej stronie niespełnienie warunku oznacza, że instrukcje w nawiasie po słowie `else` są wykonywane. Zobaczmy na przykładzie jak zachowuje się każdy z wariantów.

```
> # Wariant a: x jest typu numerycznego
> x <- 1:10
> if (is.numeric(x)) {
+     suma <- sum(x)
+     ileElem <- length(x)
+     srednia <- suma/ileElem
+ }
> srednia

[1] 5.5
```

```
> # Wariant b: x jest typu znakowego
> x <- c("a", "b", "abc")
> if (is.numeric(x)) {
+     suma <- sum(x)
+     ileElem <- length(x)
+     srednia <- suma/ileElem
+ } else {
+     srednia <- "Nie mozna obliczyc"
+    }
> srednia

[1] "Nie mozna obliczyc"
```

Można powiedzieć, że wariant b) jest bardziej elastyczny, gdyż wynikiem jego działania jest wartość numeryczna lub łańcuch znaków. Gdyby w wariancie a) przyjąć `x` typu znakowego, wtedy **R** zgłosiłby błąd.

Warunek w omówionej funkcji `if...else` może przyjmować pojedynczą wartość logiczną. Czasami zdarza się, że chcemy poddać testowi każdy element wektora. Chociaż można użyć pętli (zob. rozdz. 3.2), wygodnie jest posługiwać się funkcją:

```
ifelse(warunki, wykonaj_jesli_TRUE, wykonaj_jesli_FALSE)
```

której działanie wyjaśnia ten prosty przykład:

```
> x <- c(1,5,4,3,2,7,8,9,2,4)
> ifelse(x > 7, "wieksza", "mniejsza")

 [1] "mniejsza" "mniejsza" "mniejsza" "mniejsza" "mniejsza" "mniejsza" "wieksza"
 [8] "wieksza"  "mniejsza" "mniejsza"
```

Zauważmy, że na pierwszej pozycji mamy wektor wartości logicznych, bo **R** sprawdza kolejno, czy 1>7, czy 5>7, czy 4>7 itd. W zależności od wyniku przyjmowana jest `wieksza` (gdy `TRUE`) lub `mniejsza` (gdy `FALSE`). Należy pamiętać, że **R** niezależnie od wartości wektora logicznego i tak wykonuje ewaluacje dla obu pozycji, tj. `TRUE` i `FALSE`, co pokazuje poniższy przykład:

```
> # Oblicz pierwiastek z x lub
> # pierwiastek z wartości bezwzględnej  z x
> x <- c(9, 4, -1)
> ifelse(x >= 0, sqrt(x), sqrt(abs(x)))

Warning: NaNs produced

[1] 3 2 1
```

Jak można zauważyć, ostrzeżenie pojawia się, gdyż `sqrt(x)` obliczane jest dla wszystkich elementów (nawet -1); podobnie dzieje się w wypadku `sqrt(abs(x))`. Wywołanie funkcji gwarantuje natomiast, że zostaną wyświetlone odpowiednie wartości, zależne od wektora logicznego. W odróżnieniu od tej funkcji w wyrażeniu `if...else` obliczenia wykonywane są albo dla `TRUE` albo dla `FALSE`. Jeśli byłoby inaczej, wtedy w wariancie b) otrzymalibyśmy komunikat o błędzie. Warto zapamiętać te różnice.

3.2. Pętla for

Pętla umożliwia wielokrotne wykonywanie poleceń będących w zasięgu pętli.

```
for (i in zbioru) {
    tutaj_lista_polecen
}
```

Powyższy zapis czytamy: dla `i` należącego (`in`) do `zbioru` wykonaj wszystkie polecenia mieszczące się w nawiasie klamrowym. Czyli w pierwszej iteracji `i` przyjmuje wartość równą pierwszemu elementowi `zbioru`, w drugiej iteracji `i` ma wartość równą drugiemu elementowi `zbioru` itd.

```
> # Każdy element wektora podnieś do kwadratu i wyświetl wartość
> liczby <- c(5,3,4,-7)
> for (i in liczby) {
+     print(i^2)
+ }

[1] 25
[1] 9
[1] 16
[1] 49
```

Powyższy przykład niech służy jako ilustracja zachowania pętli, a nie jako sposób rozwiązania problemu, gdyż zdecydowanie szybciej wynik otrzymamy wpisując zamiast powyższego: `liczby^2`. I nie chodzi tu o szybkość związaną z zapisem samych instrukcji, ale szybkość z jaką **R** wykonuje obliczenia. Pętle nie są mocną stroną **R**, gdyż został on zoptymalizowany do pracy na wektorach.

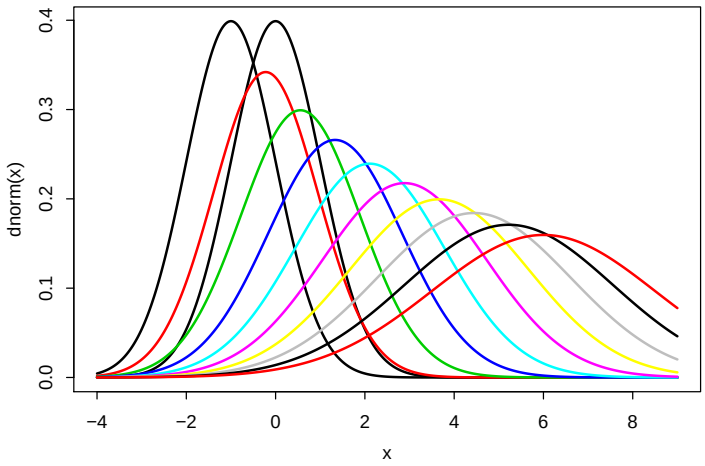
Napiszmy krótki program, który będzie obliczał sumę w każdej kolumnie macierzy, a więc realizował zadanie identyczne z poznaną już funkcją `apply()`.

```
> # Oblicz sumę w każdej kolumnie macierzy x
> x <- matrix(rnorm(50), nc=10) #Generujemy liczby i tworzymy macierz
> suma_x <- 0 #tutaj zapiszemy sumy
> for (j in 1:ncol(x)) {
+     suma_x[j] <- sum(x[, j])
+ }
> suma_x

 [1]  0.526071 -0.676366 -0.731426  1.510806 -0.691149 -2.314322  0.143808 -2.189465
 [9] -0.341123  1.991356

> apply(x, 2, sum) #to samo co wyżej

 [1]  0.526071 -0.676366 -0.731426  1.510806 -0.691149 -2.314322  0.143808 -2.189465
 [9] -0.341123  1.991356
```

Zbiór wartości jaki przyjmuje indeks `j` to $\{1,2,\ldots,10\}$, bo `ncol(x)` zwraca wartość 10. W pierwszej iteracji (`j=1`) wybieramy pierwszą kolumnę macierzy `x`, następnie sumujemy jej wartości, a powstały wynik zapisujemy jako pierwszy element wektora `suma_x`. Tę procedurę powtarzamy 10 razy.

Pętle mogą przydać się w różnych sytuacjach, oszczędzają wiele czasu. Wróćmy do przykładu ze strony 29, w którym rysowaliśmy funkcje gęstości trzech rozkładów normalnych. Wykorzystując pętlę narysujmy 10 takich gęstości.

```
> par(mar = c(4, 4, .1, .1), cex.lab = .95, cex.axis = .9, mgp = c(2, .7, 0), tcl = -.3)
> x <- seq(-4, 9, by=0.01) #przedział: jakie wartości przyjmuje x
> sr <- seq(-1, 6, length.out=10) #wektor 10 średnich
> odch <- seq(1, 2.5, length.out=10) #wektor 10 odchyleń standardowych
> plot(x, dnorm(x), type="l", lwd=2) # "l" - linia, 2 - grubość lini
> for(i in 1:length(sr)) {
+     lines(x, dnorm(x, sr[i], odch[i]), lwd=2, col=i) #col może być też liczbą
+ }
```



3.3. Funkcje

Do tej pory poznaliśmy wiele funkcji. Co ciekawe, nawet operatory są funkcjami, choć sposób ich wywołania jest odmienny. O dużej użyteczności funkcji mogliśmy się już przekonać. Szczególnie widoczne jest to wtedy, gdy takiej funkcji używamy wielokrotnie. Właśnie powtarzalność pewnych fragmentów kodu oraz ich uniwersalność czyni z funkcji narzędzie, bez którego na dłuższą metę nie można się obejść. Składnia deklaracji funkcji z trzema argumentami jest następująca:

```
nazwaFunkcji <- function(arg1, arg2, arg3) {
  cialoFunkcji
  return(jakisObiekt)
}
```

a jej wywołanie następuje wtedy, gdy napiszemy

```
  nazwaFunkcji(arg1=podaj1, arg2=podaj2, arg3=podaj3)
lub
  nazwaFunkcji(podaj1, podaj2, podaj3)
```

Funkcja posiada 3 argumenty, które muszą zostać podane, gdy funkcja jest wywoływana. W momencie wywołania wykonywany jest blok instrukcji, który stanowi ciało funkcji. Wszystkie obiekty powołane do „życia" wewnątrz funkcji mają zasięg lokalny, a więc nie są widoczne poza funkcją. Funkcja może jednak zwracać (przekazywać) wartość, jeśli wykorzystamy funkcję `return()`. Jej obiektem może być cokolwiek, np. wektor, ramka danych, lista itp. Możemy również w ostatniej linijce wpisać nazwę obiektu, który ma być zwrócony, bez używania `return()`.

W poniższym przykładzie tworzymy funkcję, która oblicza średnią z podanego wektora. Wzorcowo napisana funkcja powinna sprawdzać argumenty, aby zweryfikować, czy może być wykonana. W wypadku średniej powinniśmy sprawdzić, czy wektor jest typu numerycznego, czy liczba elementów wektora nie jest równa zero oraz co zrobić z brakami danych. Choć sprawdzaniem poprawności argumentów nie będziemy się zajmować, to jednak dla przykładu, tylko dla tej funkcji sprawdzimy typ i liczbę elementów.

```
> srednia <- function(x) {
+     #Oblicz średnią i wariancję z wartości wektora
+     stopifnot(is.numeric(x) && length(x) > 1) #Jeśli oba warunki niespełnione przerwij

+     N <- length(x)
+     mojaSrednia <- sum(x)/N
+     mojaWariancja <- (x - mojaSrednia)^2
+     mojaWariancja <- sum(mojaWariancja)/N
```

```
+    return(c(mojaSrednia, mojaWariancja)) #zwraca wektor; zamiast c, użyj data.frame
+ }
```

Wywołajmy funkcję z różnymi argumentami:

```
> myVec <- c(2,7,3,5,4,1,9)
> srednia(myVec)

[1] 4.42857 6.81633

> srednia(rnorm(20)) #średnia z wygenerowanych liczb

[1] 0.286412 1.061766

> srednia(c("a", "b")) #nie jest numeryczny, więc błąd

Error: is.numeric(x) && length(x) > 1 is not TRUE
```

3.4. Zadania

Zad. 1. Wykorzystaj poniższy fragment do wygenerowania ocen

```
> oceny <- sample(c(2:5, 3.5, 4.5), 100, replace=TRUE)
```

Następnie stwórz wektor, którego elementami będzie informacja: pozytywna (gdy ocena przynajmniej 3) lub negatywna (gdy 2). Chętni mogą spróbować rozszerzyć przykład na sytuację, w której elementami wektora będzie słowna nazwa oceny, np. gdy 5 to `b.dobry`.

Zad. 2. Napisać funkcję `WspolRozklad`, która dla wektora numerycznego, o minimalnej długości 4, zwróci wartości dwóch charakterystyk rozkładu: współczynnik asymetrii (skośność) i współczynnik spłaszczenia (kurtoza). Do obliczeń należy wykorzystać poniższe wzory na ich oszacowanie:

$$sk = \frac{N}{(N-2)(N-1)} \frac{\sum_{i=1}^{N}(x_i - \bar{x})}{s^3}$$

$$k = \frac{n(n+1)}{(n-1)(n-2)(n-3)} \frac{\sum_{i=1}^{N}(x_i - \bar{x})^4}{s^4} - 3\frac{(n-1)^2}{(n-2) * (n-3)}$$

$$\text{gdzie} \quad s^2 = \frac{1}{N-1}\sum_{i=1}^{N}(x - \bar{x})^2$$

Funkcje w **R**, które obliczają s^2 oraz s według podanego wyżej wzoru to odpowiednio: `var()` oraz `sd()`. Do obsługi błędów można wykorzystać:

```
stopifnot(is.numeric(x) && length(x) > 3)
```

lub

```
if (!is.numeric(x) || length(x) < 4){
    warning("argument nie jest numeryczny lub liczba elementow < 4: zwraca
        NA")
    return(NA)
}
```

Poprawność zweryfikować z poniższym przykładem.

```
> set.seed(123) #ustaw ziarno generatora
> x <- rchisq(100, 2) #generujemy dane
> WspolRozklad(x)
```

```
  asymetria splaszczenie
1   1.92805      4.74577
> WspolRozklad(c(1,4,2)) # wektor ma mniejszą długość niż 4
Warning: argument nie jest numeryczny lub liczba elementow < 4: zwraca NA
[1] NA
```

Rozdział 4

Przygotowanie danych do analizy

Aby zapisać lub odczytać plik, **R** musi znać ścieżkę dostępu. Gdy jej nie podamy, stosowana jest domyślna lokalizacja. Pracując z programem RStudio, w systemie projektów (zob. rozdz. 1.1) tą domyślną lokalizacją jest folder projektu. Możemy się o tym przekonać, wywołując funkcję `getwd()`, która jest skrótem od: *get working directory* (weź katalog roboczy)[1].

Nic nie stoi na przeszkodzie, aby wszystkie pliki zapisywać w domyślnym folderze (katalogu roboczym). Dobrą praktyką jest, przynajmniej w wypadku rozbudowanych analiz, zapisywanie zbiorów danych, plików graficznych (wykresy) itp. w osobnych katalogach. Kierując się tą zasadą, od tej pory, wszystkie dane wczytywane/zapisywane będą w folderze *dane*. Oznacza to, że zamiast samej nazwy pliku (np. `mojplik.txt`) dodatkowo wskazywać będziemy na folder z danymi (`dane/mojplik.txt`). Oczywiście katalog *dane* należy uprzednio utworzyć. Można to zrobić w znany każdemu sposób, albo wywołać funkcję `dir.create("dane")`.

W systemie Windows jako separatora katalogów używamy \ tzw. odwróconego ukośnika (*backslash*). Ponieważ **R** go nie akceptuje, dlatego należy go zastąpić \\ albo /. Ten ostatni zgodny jest z systemami Linux oraz Mac, więc będziemy go używać.

4.1. Wczytywanie i zapisywanie danych

Wczytywanie danych

Najczęściej zbiory danych, które chcemy poddać analizie, znajdują się w plikach zewnętrznych. Wczytanie danych do **R** mających strukturę tabeli realizuje funkcja `read.table()`. Ma ona dość rozbudowaną listę argumentów, dlatego ograniczymy się tylko do wybranych, których manipulacja w zdecydowanej większości wypadków jest wystarczająca. Ogólna postać tej funkcji, z domyślnymi argumentami, jest następująca:

```
read.table("nazwa_pliku", header = FALSE, sep = "", dec = ".",
           na.strings = "NA", encoding = "unknown")
```

`header`	—	czy pierwszy wiersz zawiera nazwy zmiennych: tak – `TRUE`, nie – `FALSE`;
`sep`	—	jaki jest znak separacji kolumn; domyślne `""` oznacza dowolny biały znak (spacja, tabulator); wpisując `sep="\t"` informujmy *explicite* o znaku tabulacji;
`dec`	—	znak separacji części całkowitych i dziesiętnych; jeśli liczby są postaci 2,35 wtedy `dec = ","`;
`na.strings`	—	w jaki sposób oznaczone są braki danych; domyślnie `NA`;
`encoding`	—	jakie jest kodowanie znaków; najlepiej jej nie używać, ale jeśli po wczytaniu nie ma polskich znaków można wybrać `encoding="UTF-8"`; użytkownicy systemu Windows mogą również rozważyć kodowanie `CP1250`.

[1]Jeżeli nie pracujemy w systemie projektów lub uruchomimy program **R** (nie RStudio), wtedy domyślnym katalogiem roboczym w systemie Windows jest katalog *Moje dokumenty*.

Widoczne po znaku = wartości traktowane są jako wartości domyślne, co oznacza, że jeśli np. z powyższego zapisu usuniemy `header=FALSE`, wtedy **R** i tak ustawi `header` na wartość `TRUE`. Tę funkcję będziemy wykorzystywać do wczytywania plików rozszerzeniami: *txt* oraz *dat*.

Mając pliki z rozszerzeniem *csv* — w których rolę separatora części dziesiętnych pełni przecinek, a separatora kolumn średnik — wygodnie jest posłużyć się funkcją `read.csv2()`. Można powiedzieć, że jest to funkcja `read.table()` z ustawionymi odpowiednio wartościami argumentów[2]. Porównajmy poniższe dwa, równoważne zapisy (zapewne każdy wybierze ten krótszy):

```
read.csv2("mojPlik.csv")
read.table("mojPlik.csv", header = TRUE, sep = ";", dec = ",")
```

Popularność plików z rozszerzeniami *xls, xlsx* (tzw. plików Excela) sprawia, że chcielibyśmy umieć dane w tych formatach wczytywać do **R**. Chociaż istnieją narzędzia, jak np. rewelacyjny pakiet `XLConnect`, to jednak kierując się prostotą, proponuję zapisać taki plik w formacie[3] *csv*, a następnie wykorzystać funkcję `read.csv2()`.

Poniższy przykład ilustruje zachowanie omówionych funkcji, w których uwzględniono minimalną liczbę niezbędnych argumentów. Pominiecie któregoś nie pozwoli poprawnie zaimportować danych, np. pominiecie `na.strings=-1` spowoduje, że -1 będzie traktowane jako liczba, a nie jako zakodowany brak danych. Zwróćmy jeszcze uwagę na to, że katalog w którym znajdują się wczytywane poniżej pliki to *dane*. Należy go utworzyć i skopiować tam pliki.

```
> powiaty <- read.table("dane/powiaty.txt", header=TRUE, dec=",")
> usa <- read.table("dane/P081.dat", header=TRUE)
> zatrud <- read.table("dane/zatrudnienie.dat", header=TRUE, dec=",", na.strings=-1)
> zatrud0 <- read.csv2("dane/zatrudnienie0.csv", encoding = "UTF-8")
```

Zapisywanie danych do pliku

Ramki danych możemy zapisać korzystając z funkcji `write.table()`. Jej postać wraz z wybranymi wartościami argumentów domyślnych jest następująca:

```
write.table(x, file = "", quote = TRUE, sep = " ", na = "NA", dec = ".",
            row.names = TRUE, col.names = TRUE)
```

`x`	—	nazwa ramki danych, którą chcemy zapisać;
`file`	—	ujęta w cudzysłowy nazwa pliku wraz z rozszerzeniem;
`quote`	—	czy zmienne typu znakowego ująć w cudzysłowy;
`sep`	—	domyślnym separatorem kolumn jest spacja; jeśli `quote = FALSE`, wtedy za separator lepiej przyjąć coś innego[4], np. tabulator (`"\t"`);
`na`	—	jak mają być zakodowane braki danych;
`row.names`	—	czy nazwy wierszy mają być zapisane;
`col.names`	—	czy nazwy kolumn mają być zapisane; tutaj warto ustawić na `FALSE`.

Funkcja `write.csv2()` stanowi alternatywę dla powyższej, gdy chcemy plik zapisać w formacie *csv*, ze wszystkimi tego konsekwencjami opisanymi w części o wczytywaniu danych.

Zobaczmy, jak wyglądają pliki danych, jeśli użyjemy do ich utworzenia różnych funkcji lub/i różnych argumentów. Najpierw stwórzmy ramkę danych.

```
set.seed(123)
opinia <- sample(c("Zdecydowanie nie", "Raczej nie", "Ani tak, ani nie", "Raczej tak",
                   "Zdecydowanie tak"), 100, replace=TRUE)
wykszt <- sample(c("zawodowe lub niższe", "średnie", "wyższe"), 100, TRUE)
wynagro <- round(rnorm(100, 3000, 1000), 2)
```

[2] Inne tzw. nakładki na funkcję bazową to `read.csv()`, `read.delim()`, `read.delim2()`

[3] W tym celu z poziomu Excela wybieramy *zapisz jako*, a następnie z rozwijanej listy *zapisz jako typ* wskazujemy na *CSV (rozdzielany przecinkami)*.

```
dfbad <- data.frame(opinia, wykszt,  wynagro)
head(dfbad, 5)

            opinia                wykszt wynagro
1       Raczej nie              średnie 2289.59
2       Raczej tak zawodowe lub niższe 3256.88
3 Ani tak, ani nie              średnie 2753.31
4 Zdecydowanie tak               wyższe 2652.46
5 Zdecydowanie tak              średnie 2048.38
```

Zapiszmy wygenerowane dane na kilka alternatywnych sposobów.

```
> write.table(dfbad, "dane/badanie1.txt")
> write.table(dfbad, "dane/badanie2.txt", sep="\t", quote=FALSE, row.names=FALSE)
> write.table(dfbad, "dane/badanie3.txt", dec=",", sep="\t", quote=FALSE, row.names=F)
> write.csv2(dfbad, "dane/badanie4.csv", row.names=FALSE)
> write.csv2(dfbad, "dane/badanie5.csv", row.names=FALSE, quote=FALSE)
```

Po zapisaniu, pliki można otworzyć w dowolnym edytorze tekstu (np. *Notepad++*) i zobaczyć efekty. Proponuję wykonać to ćwiczenie, analizując różnice. Poniżej zamieszczam zrzuty do porównania.

badanie1.txt

```
"opinia" "wykszt" "wynagro"
"1" "Raczej nie" "średnie" 2289.59
"2" "Raczej tak" "zawodowe lub niższe" 3256.88
"3" "Ani tak, ani nie" "średnie" 2753.31
"4" "Zdecydowanie tak" "wyższe" 2652.46
```

badanie2.txt

```
opinia  wykszt  wynagro
Raczej nie  średnie 2289.59
Raczej tak  zawodowe lub niższe 3256.88
Ani tak, ani nie    średnie 2753.31
Zdecydowanie tak    wyższe  2652.46
```

badanie3.txt

```
opinia  wykszt  wynagro
Raczej nie  średnie 2289,59
Raczej tak  zawodowe lub niższe 3256,88
Ani tak, ani nie    średnie 2753,31
Zdecydowanie tak    wyższe  2652,46
```

badanie4.csv

```
"opinia";"wykszt";"wynagro"
"Raczej nie";"średnie";2289,59
"Raczej tak";"zawodowe lub niższe";3256,88
"Ani tak, ani nie";"średnie";2753,31
"Zdecydowanie tak";"wyższe";2652,46
```

badanie5.csv

```
opinia;wykszt;wynagro
Raczej nie;średnie;2289,59
Raczej tak;zawodowe lub niższe;3256,88
Ani tak, ani nie;średnie;2753,31
Zdecydowanie tak;wyższe;2652,46
```

4.2. Wybór przypadków i zmiennych do analizy

Dość często analizy wykonujemy nie na całym zbiorze danych ale pewnym jego podzbiorze. Wyboru przypadków (obserwacji/wierszy) lub/i zmiennych można dokonać wykorzystując znany już sposób (zob. str. 23) oparty na operatorze `[`. Nierzadko wygodniej jest użyć odpowiedniej funkcji, która sprawia, że zapis jest bardziej czytelny. W pakiecie podstawowym (*base*) **R** mamy funkcję

```
subset(x, subset, select)
```

której parametrami są: `x` — nazwa zbioru danych, `subset` — wyrażenie logiczne wskazujące, które przypadki wybrać, `select` — wyrażenie wskazujące na zmienne, które należy wybrać. Jak zobaczymy później (w przykładzie ilustrującym), nazwę parametru `subset` zazwyczaj pomijamy, natomiast w wypadku `select` piszemy: `select=nazwy_kolumn`.

Omówimy jeszcze dwie funkcje: `filter()` i `select()` z pakietu `dplyr`, których funkcjonalność jest identyczna z `subset()`. Można się zastanawiać, po co wprowadzać kolejne. Otóż w rozdziale dotyczącym eksploracyjnej analizy danych będziemy ze wspomnianego pakietu korzystać. Jak się przekonamy, zaimplementowana tam filozofia i podejście do przetwarzania danych czynią ten proces bardzo

przejrzystym i dość prostym. Zamiast wprowadzać je później, warto już teraz dokonać równoległych porównań z funkcją `subset()`.

W założeniu twórców pakietu `dplyr`, każda funkcja wykonuje ściśle określone zadanie — dlatego potrzebujemy dwóch funkcji, aby uzyskać funkcjonalności równoważną z funkcją `subset()`. Za wybór przypadków odpowiada funkcja `filter()`, a za wybór zmiennych funkcja `select()`:

```
filter(x, warunek_1, warunek_2, ...)
select(x, zmienna_1, zmienna_2, ...)
```

Wpisywane po przecinku warunki traktowane są jak koniunkcja warunków — wszystkie muszą być spełnione. Gdy interesuje nas alternatywa, wtedy wystarczy użyć operatora `|`. Należy zapamiętać, że jednoczesny wybór przypadków i zmiennych odbywa się w dwóch krokach: w pierwszym wykorzystujemy jedną z funkcji, by w następnym użyć drugiej, której argumentem `x` jest wynik wywołania pierwszej funkcji. Kolejność użycia funkcji nie ma znaczenia.

W tym miejscu powinniśmy wspomnieć o operatorze `%in%`, który oznacza: *należy* i zastępuje matematyczny symbol $\in$. Jest on szczególnie użyteczny wtedy, gdy chcemy, aby wartości zmiennej należały do zbioru, np. `x %in% c("zielony", "czerwony")`.

Poniżej przedstawimy przykład ilustrujący zachowanie różnych podejść do wyboru podzbioru danych. Każdy w ramach przedstawionego problemu jest równoważny, dlatego rezultaty zostały wyświetlone dla ostatniego podejścia. Celowo więc wyniki pierwszych podejść zostały zapisane jako obiekty `y1`, `y2` itd., aby **R** ich nie wyświetlał. Jak zwykle potrzebujemy zbioru danych, który i tym razem wygenerujemy przy użyciu poniższego kodu.

```
## Generujemy wektory do ramki danych
ileObser <- 20
set.seed(12345)
plec <- sample(c("k", "m"), ileObser, replace=TRUE, prob=c(0.7,0.3))
wiek <- sample(c(20:60), ileObser, replace=TRUE)
mieszka <- sample(c("miasto", "wies"), ileObser, replace=TRUE, prob=c(0.7,0.3))
papierosy <- sample(0:10, ileObser, replace=TRUE, prob=c(0.9, rep(0.2, times=10)))
wwwGodziny <- sample(0:15, ileObser, replace=TRUE)
```

Możemy teraz wykorzystać wygenerowane wektory danych do stworzenie ramki danych[5]. Nazwa zbioru danych powinna mówić coś o samych danych. Kierując się celem dydaktycznym — łatwiej dostrzec różnice przy krótszych nazwach — nich `x` będzie nazwą nowego obiektu **R**.

```
> ## Tworzymy ramkę danych:
> x <- data.frame(plec, wiek, mieszka, papierosy, wwwGodziny)
> x

   plec wiek mieszka papierosy wwwGodziny
1     m   38    wies         8         13
2     m   33  miasto         0          8
3     m   59    wies         1          0
4     m   49    wies         8          0
5     k   46  miasto         1          2
6     k   35  miasto         0          4
7     k   48  miasto         1         13
8     k   42  miasto         0          8
9     m   29  miasto         6         12
10    m   39  miasto         1          0
11    k   52    wies         3         14
12    k   20    wies         4         12
13    m   27  miasto         2          1
14    k   47  miasto         2          9
```

[5]Czy mogę stworzyć macierz z tych wektorów?

```
15    k   35    wies         0         11
16    k   34  miasto         6          8
17    k   55    wies         1         11
18    k   57  miasto         6         11
19    k   45  miasto         4          1
20    m   25  miasto         0          6
```

Wybór przypadków

1. Wybrać tylko mężczyzn

```
> library(dplyr) #aby użyć funkcji z pakietu dplyr musimy go wczytać
> y1 <- x[x[ ,1] == "m", ] #sposób 1 lub: x[x[1] == "m", ]
> y2 <- x[x[ ,"plec"] == "m", ] #sposób 2 lub: x[x["plec"] == "m", ]
> y3 <- subset(x, plec == "m") #sposób 3
> filter(x, plec == "m") #sposób 4

  plec wiek mieszka papierosy wwwGodziny
1    m   38    wies         8         13
2    m   33  miasto         0          8
3    m   59    wies         1          0
4    m   49    wies         8          0
5    m   29  miasto         6         12
6    m   39  miasto         1          0
7    m   27  miasto         2          1
8    m   25  miasto         0          6
```

2. Wybrać tych, którzy wypalają więcej niż 5 papierosów dziennie

```
> y1 <- x[x[ ,4] > 5, ] #sposób 1
> y2 <- x[x[ ,"papierosy"] > 5, ] #sposób 2
> y3<- subset(x, papierosy > 5) #sposób 3
> filter(x, papierosy > 5) #sposób 4

  plec wiek mieszka papierosy wwwGodziny
1    m   38    wies         8         13
2    m   49    wies         8          0
3    m   29  miasto         6         12
4    k   34  miasto         6          8
5    k   57  miasto         6         11
```

3. Wyłączyć z analizy tych, którzy wypalają 0 lub 1 papierosa dziennie

```
> y1 <- x[x[ ,4] != 0 & x[ ,4] != 1, ] #sposób 1
> y2 <- x[x[ ,"papierosy"] != 0 & x[ ,"papierosy"] != 1, ] #sposób 2
> y3 <- subset(x, papierosy != 0 & papierosy != 1) #sposób 3
> y4 <- filter(x, !papierosy %in% c(0,1))  #sposob 4a: ! - oznacza: nieprawda
> filter(x, papierosy != 0, papierosy != 1) #sposób 4b

   plec wiek mieszka papierosy wwwGodziny
1     m   38    wies         8         13
2     m   49    wies         8          0
3     m   29  miasto         6         12
4     k   52    wies         3         14
```

```
5     k   20    wies         4         12
6     m   27  miasto         2          1
7     k   47  miasto         2          9
8     k   34  miasto         6          8
9     k   57  miasto         6         11
10    k   45  miasto         4          1
```

4. Wybrać: niepalących, którzy spędzają przed internetem przynajmniej 8 godzin

```
> y1 <- x[x[ ,4] == 0 & x[ ,5] >= 8, ] #sposób 1
> y2 <- x[x[ ,"papierosy"] == 0 & x[ ,"wwwGodziny"] >= 8, ] #sposób 2
> y3<- subset(x, papierosy == 0 & wwwGodziny >= 8) #sposób 3
> filter(x, papierosy == 0, wwwGodziny >= 8) #sposób 4

  plec wiek mieszka papierosy wwwGodziny
1    m   33  miasto         0          8
2    k   42  miasto         0          8
3    k   35    wies         0         11
```

5. Wybrać niepalące kobiety ze wsi

```
> y1 <- x[x[ ,1] == "k" & x[ ,3] == "wies" & x[ ,4] == 0, ] #sposób 1
> y2 <- x[x[ ,"plec"] == "k" & x[ ,"mieszka"] == "wies" & x[ ,"papierosy"] == 0, ]
> y3 <- subset(x, plec == "k" & mieszka == "wies" & papierosy == 0) #sposób 3
> filter(x, plec == "k", mieszka == "wies", papierosy == 0) #sposób 4

  plec wiek mieszka papierosy wwwGodziny
1    k   35    wies         0         11
```

6. Wyłączyć osoby miedzy 30 a 50 rokiem życia

```
> y1 <- x[x[ ,2] > 50 | x[ ,2] < 30, ] #sposób 1
> y2 <- x[x[ ,"wiek"] > 50 | x[ ,"wiek"] < 30 , ] #sposób 2
> y3 <- subset(x, wiek > 50 | wiek < 30) #sposób 3
> filter(x, wiek > 50 | wiek < 30) #sposób 4

  plec wiek mieszka papierosy wwwGodziny
1    m   59    wies         1          0
2    m   29  miasto         6         12
3    k   52    wies         3         14
4    k   20    wies         4         12
5    m   27  miasto         2          1
6    k   55    wies         1         11
7    k   57  miasto         6         11
8    m   25  miasto         0          6
```

Wybór zmiennych

1. Wybrać zmienne: `wiek, papierosy` *i* `wwwGodziny`

```
> y1 <- x[ ,c(2, 4, 5)]
> y2 <- x[, c("wiek", "papierosy", "wwwGodziny")]
> y3 <- subset(x,  select = c(wiek, papierosy, wwwGodziny))
> y4 <- select(x, wiek, papierosy, wwwGodziny)
> head(y4, 2) #wyświetl tylko 2 wiersze
```

```
  wiek papierosy wwwGodziny
1   38         8         13
2   33         0          8
```

2. Wybrac wszystkie zmienne: od `wiek` *do* `papierosy`

```
> y1 <- x[ ,2:4]
> y2 <- subset(x, select = wiek:papierosy)
> y3 <- select(x, wiek:papierosy)
> head(y3, 2) #wyświetl tylko 2 wiersze

  wiek mieszka papierosy
1   38    wies         8
2   33  miasto         0
```

3. Wybrać zmienna pierwsza, oraz zmienne od `mieszka` *do* `wwwGodziny`

```
> y1 <- x[ ,c(1, 3:5)]
> y2 <- subset(x, select = c(plec, mieszka:wwwGodziny))
> y3 <- select(x, plec, mieszka:wwwGodziny)
> head(y3, 2) #wyświetl tylko 2 wiersze

  plec mieszka papierosy wwwGodziny
1    m    wies         8         13
2    m  miasto         0          8
```

Wybór przypadków i zmiennych

Wybrać zmienne `plec` *i* `papierosy` *oraz te respondentki, które wypalają mniej niż 5 papierosów*

```
> y1 <- x[x[ ,1] == "k" & x[ ,4] < 5, c(1, 4)]
> y2 <- subset(x,  plec == "k" & papierosy < 5, select = c(plec, papierosy))
> krok1 <- select(x, plec, papierosy) #najpierw zmienne (zobacz jak wygląda krok1)
> krok2 <- filter(krok1, plec == "k", papierosy < 5) #później przypadki
> krok2

   plec papierosy
1     k         1
2     k         0
3     k         1
4     k         0
5     k         3
6     k         4
7     k         2
8     k         0
9     k         1
10    k         4
```

Zauważmy, że w powyższym przykładzie najpierw użyliśmy funkcji `select()`, by w kolejnym kroku wynik tej operacji poddać następnemu przekształceniu, czyli działaniu funkcji `filter()`. Oczywiście nic nie stoi na przeszkodzie, aby w jednym kroku otrzymać wynik, zapisując:

```
select(filter(krok1, plec == "k", papierosy < 5), plec, papierosy)
```

Takie podejście może być nieczytelne, dlatego gorąco zachęcam do korzystania z operatora %>%, którego działanie wyjaśnia przykład: wyrażenie `x %>% f(y)` jest równoważne z zapisem `f(x,y)`. Możemy powiedzieć: `x` ma być pierwszym argumentem funkcji `f(y)`, czyli `f(x,y)`. Stosując takie podejście zapiszemy (sprawdź, czy poniższy obiekt `krok2` jest identyczny z wynikiem poprzednim):

```
> krok2 <- x %>%
+           select(plec, papierosy) %>%
+           filter(plec == "k", papierosy < 5)
```

Najpierw `x` podstawiany jest na pierwsze miejsce do funkcji `select()`, by następnie wynik tej operacji wstawić na pierwsze miejsce do funkcji `filter()`.

4.3. Przekształcanie zmiennych, sortowanie

Sortowanie względem zmiennych

Sortowanie ramek danych najwygodniej jest wykonać przy użyciu funkcji `arrange()` będącej częścią pakietu `dplyr`. Jeśli napiszemy: `arrange(x, zmienna1, zmienna2, desc(zmienna3))`, wtedy zbiór danych `x` zostanie posortowany najpierw ze względu na zmienną 1, później ze względu na zmienną 2 i ostatecznie w porządku malejącym (*descending*) ze względu na zmienną 3. Jeżeli nie pojawi się `desc()`, wtedy domyślnie sortowanie odbywa się w porządku rosnącym.

Wykorzystajmy wygenerowane dane z rozdz. 4.2, do posortowania ich względem zmiennych: `plec`, `miasto`, `papierosy` przy czym `miasto` ma być posortowane w porządku malejącym (czyli `wies` będzie pierwsza):

```
> arrange(x, plec, desc(mieszka), papierosy)

   plec wiek mieszka papierosy wwwGodziny
1     k   35    wies         0         11
2     k   55    wies         1         11
3     k   52    wies         3         14
4     k   20    wies         4         12
5     k   35  miasto         0          4
6     k   42  miasto         0          8
7     k   46  miasto         1          2
8     k   48  miasto         1         13
9     k   47  miasto         2          9
10    k   45  miasto         4          1
11    k   34  miasto         6          8
12    k   57  miasto         6         11
13    m   59    wies         1          0
14    m   38    wies         8         13
15    m   49    wies         8          0
16    m   33  miasto         0          8
17    m   25  miasto         0          6
18    m   39  miasto         1          0
19    m   27  miasto         2          1
20    m   29  miasto         6         12
```

Można również skorzystać z wbudowanej funkcji `order()`. Aby osiągnąć przybliżony efekt z przykładu, bez malejącego porządku dla zmiennej `miast`, należałoby napisać:

```
x[order(x$plec, x$mieszka, x$papierosy), ]
```

Jest to zapewne mniej czytelne, a w wypadku sortowania zmiennych według różnych porządków (rosnących, malejących) dość kłopotliwe.

Przekształcenia zmiennych

Przekształcanie zmiennej polega na jej transformacji, której wynik zapisujemy pod inną, bądź taką samą nazwą (wtedy nadpisujemy oryginalną zmienną). Wynik całej operacji, gdy tworzymy nową zmienną, powiększa nam ramkę danych o dodatkową kolumnę. Bardzo łatwa w użyciu i zarazem czytelna w zapisie jest funkcja `mutate()` z pakietu `dplyr`. Zilustrujmy jej zachowanie na przykładzie[6].

```
> # Tworzymy ramkę danych: badanie
> pracownik <- c("kierownik", "wykonawczy", "wykonawczy", "kierownik")
> rokUrodz <- c(1962, 1975, 1990, 1959)
> wynUSD <- c(57000, 40200, 21450, 21900)
> wynPoczUSD <- c(27000, 18750, 12000, 13200)
> badanie <- data.frame(pracownik, rokUrodz, wynUSD, wynPoczUSD)
> badanie

   pracownik rokUrodz wynUSD wynPoczUSD
1  kierownik     1962  57000      27000
2 wykonawczy     1975  40200      18750
3 wykonawczy     1990  21450      12000
4  kierownik     1959  21900      13200
```

W oparciu o ramkę danych `badanie` przeprowadzić następujące przekształcenia, a powstałą w ten sposób ramkę, zapisać jako nowy obiekt pod nazwą `badanieNew`:

- Wykorzystując rok urodzenia (`rokUrodz`) utworzyć zmienną `wiek`.
- Utworzyć zmienną `wynagroDiffUSD`, której wartości są różnicą między obecnym wynagrodzeniem (`wynagoUSD`) a wynagrodzeniem początkowym (`wynagoPoczUSD`)
- Zmienną utworzoną w poprzednim punkcie wyrazić w PLN — utworzyć zmienną `wynagroDiffPLN` przyjmując kurs wymiany na poziomie 3.19 PLN/USD

```
> badanieNew <- mutate(badanie, wiek = 2014 - rokUrodz,
+                              wynDiffUSD = wynUSD - wynPoczUSD,
+                              wynDiffPLN = 3.19 * wynDiffUSD)
> badanieNew

   pracownik rokUrodz wynUSD wynPoczUSD wiek wynDiffUSD wynDiffPLN
1  kierownik     1962  57000      27000   52      30000    95700.0
2 wykonawczy     1975  40200      18750   39      21450    68425.5
3 wykonawczy     1990  21450      12000   24       9450    30145.5
4  kierownik     1959  21900      13200   55       8700    27753.0
```

Zdobyta dotychczas wiedza pozwala na wykonanie powyższych przekształceń bez użycia funkcji `mutate()` — proponują wykonać to jako ćwiczenie. Jednak prostota i klarowność zapisu sprawiają, że warto jej używać. Poniżej podaję rozwiązanie tego krótkiego ćwiczenia.

```
> wiek <- 2014 - badanie$rokUrodz
> wynDiffUSD <- badanie$wynUSD - badanie$wynPoczUSD
> wynDiffPLN <-  3.19 * wynDiffUSD
> cbind(badanie, wiek, wynDiffUSD, wynDiffPLN) #funkcja omówiona w rozdz. o macierzach

   pracownik rokUrodz wynUSD wynPoczUSD wiek wynDiffUSD wynDiffPLN
1  kierownik     1962  57000      27000   52      30000    95700.0
2 wykonawczy     1975  40200      18750   39      21450    68425.5
3 wykonawczy     1990  21450      12000   24       9450    30145.5
4  kierownik     1959  21900      13200   55       8700    27753.0
```

[6]Identyczną składnię ma funkcja `transform()` z pakietu podstawowego (`base`). Przewaga prezentowanej w przykładzie polega na tym, że możemy tworzyć zmienne będące funkcją aktualnie tworzonych. W prezentowanym przykładzie zastąp `mutate()` funkcją `transform()`. Widzimy błąd, bo tworzona zmienna jest niedostępna.

4.4. Rekodowanie zmiennych

Zmienną płeć możemy zakodować na kilka sposobów. Pierwszy — przyjąć: `k` i `m`; dugi — użyć wartości numerycznych, np. `0` i `1` (należy jednak pamiętać, która wartość oznacza mężczyzn, a która kobiety); trzeci — wziąć pełne nazwy: `kobieta` i `mężczyzna`. Mając w zbiorze danych któryś ze sposobów kodowania, możemy chcieć go zmienić. Właśnie rekodowanie, jak sama nazwa wskazuje, polega na zmianie sposobu kodowania zmiennej.

Niemal każdą operację w **R** można przyprowadzić na wiele różnych sposobów — nie inaczej jest z rekodowaniem. Wiele czasu możemy zaoszczędzić korzystając z pakietu `car` i funkcji:

```
recode(zmienna, "obecny_1='nowy_1'; obecny_2='nowy_2'; else='nowy_3'")
```

w której wartości `obecny` chcemy rekodować (zamienić) na `nowy`. Zauważmy, że całość ujęta jest w cudzysłowy "; łańcuch znaków (tutaj `nowy`) trzeba ująć w ', natomiast wartości numeryczne tego nie wymagają (`obecny`). Ostatnie kodowanie z użyciem słowa `else` oznacza: wszystkie pozostałe wartości nieujęte wcześniej mają wartość `nowy_3`.

Przeprowadźmy rekodowanie dwóch wektorów:

```
> # Tworzymy wektory: plec i miesiac
> set.seed(123)
> (plec <- sample(c("k", "m"), 10, replace=TRUE))

 [1] "k" "m" "k" "m" "m" "k" "m" "m" "m" "k"

> (miesiac <- sample(1:12, 30, replace=TRUE))

 [1] 12  6  9  7  2 11  3  1  4 12 11  9  8 12  8  9  7  8  4  2 12 11  9 10  1  6 10  3
[29]  4  3
```

Pierwszy (`plec`) ma wartości typu znakowego, drugi (`miesiac`) typu numerycznego. Schemat rekodowania wygląda następująco: skróty `k` i `m` rozwinąć do pełnych nazw, a miesiące rekodować na kwartały, tzn. miesiąc od 1 do 3 ma być zastąpiony: `I kw.` itd. Zwróć uwagę na znaki: " oraz '.

```
> library(car) #pakiet,  w którym jest funkcja recode
> recode(plec, "'k'='kobieta'; 'm'='mężczyzna'")

 [1] "kobieta"     "mężczyzna" "kobieta"     "mężczyzna" "mężczyzna" "kobieta"
 [7] "mężczyzna" "mężczyzna" "mężczyzna" "kobieta"

> recode(miesiac, "1:3='I kw.'; 4:6='II kw.'; 7:9='III kw.'; 10:12='IV kw.'")

 [1] "IV kw."  "II kw."  "III kw." "III kw." "I kw."   "IV kw."  "I kw."   "I kw."
 [9] "II kw."  "IV kw."  "IV kw."  "III kw." "III kw." "IV kw."  "III kw." "III kw."
[17] "III kw." "III kw." "II kw."  "I kw."   "IV kw."  "IV kw."  "III kw." "IV kw."
[25] "I kw."   "II kw."  "IV kw."  "I kw."   "II kw."  "I kw."
```

Zobaczmy, w jaki sposób można zmienną `zarobki` rekodować na zmienną o trzech kategoriach opisujących wysokość zarobków tj.: `niskie`, `srednie`, `wysokie`. Osobną kwestią jest rozstrzygnięcie, z jakiego przedziału zarobki należy uważać np. za niskie. Tego nie będziemy tutaj rozpatrywać, a przyjęte przedziały posłużą ilustracji. Zwróćmy uwagę na wartość specjalną `lo`, oznaczającą wartość najmniejszą, oraz słowo `else`. Dla oznaczenia wartości największej można użyć `hi`.

```
> set.seed(123)
> zarobki <- rnorm(100, mean=2200, sd=1300)
> zarKat3 <- recode(zarobki, "lo:1500='niskie'; 1500:3000='średnie'; else='wysokie'")
> table(zarKat3) #utwórz tabelę liczebności
```

```
zarKat3
średnie   niskie  wysokie
      49       24       27
```

4.5. Zadania

Zadania podstawowe

Zad. 1. Wczytaj plik z danymi `saty.dat`, a wczytany zbiór nazwij `saty`. Wyświetl nazwy zmiennych, podaj liczbę zmiennych i obserwacji, wyświetl pierwsze 10 obserwacji. Oczywiście użyć należy innej funkcji, niż `str()`. Poniżej struktura danych.

```
> str(saty)

'data.frame': 236 obs. of  14 variables:
 $ wiek           : int  25 47 23 57 21 45 28 53 34 38 ...
 $ naukaLata      : int  14 12 12 16 10 13 15 17 12 13 ...
 $ edukacja       : Factor w/ 5 levels "Licencjat","Magisterium",..: 4 4 4 1 4 4 4 2 4 4 ...
 $ plec           : Factor w/ 2 levels "Kobieta","Mezczyzna": 2 1 1 1 1 1 1 2 2 2 ...
 $ wiaraZyciePo   : Factor w/ 2 levels "Nie","Tak": 2 2 2 1 2 2 2 2 2 2 ...
 $ szczescie      : Factor w/ 3 levels "Bardzo szczesliwy",..: 2 2 2 1 1 1 2 2 2 2 ...
 $ jakieZycie     : Factor w/ 3 levels "Ekscytujace",..: 1 1 3 1 1 1 3 3 3 1 ...
 $ fortunaEndPraca: Factor w/ 2 levels "Nie","Tak": 1 2 2 1 1 1 1 2 1 2 ...
 $ gazetyCzesto   : Factor w/ 5 levels "Codziennie","Kilka razy w tygodniu",..: 1 1 2 1 2 2 1 2 1 1
 $ ogladaTVGodz   : int  1 1 6 1 3 2 1 1 4 2 ...
 $ korzystaWWW    : Factor w/ 2 levels "Nie","Tak": 2 1 1 2 2 2 1 2 1 2 ...
 $ zodiak         : Factor w/ 12 levels "Baran","Bliznieta",..: 11 8 5 6 12 5 2 1 2 1 ...
 $ dochod         : Factor w/ 21 levels "$1 000 TO 2 999",..: 11 20 12 13 13 16 18 16 6 15 ...
 $ ileGodzPracuje : int  8 10 5 9 7 8 8 9 8 12 ...
```

Zad. 2. Wybrać te zmienne, które odnoszą się do edukacji.

Zad. 3. Posortować zbiór ze względu na edukację (rosnąco) i liczbę spędzonych godzin przed TV (malejąco).

Zad. 4. Wybrać osoby, które wierzą w życie po śmierci i są spod znaku lwa.

Zad. 5. Wybrać te osoby, których znak zodiaku zaczyna się od litery B.

Zad. 6. Wziąć wszystkie zmienne od *wieku* do *płci* włącznie, a następnie z tego zbioru wybrać osoby, które uczyły się więcej niż 19 lat.

Zad. 7. W zbiorze danych są zmienne, które mają charakter porządkowy (np. częstotliwość czytania gazet). Choć są reprezentowane jako czynniki, to jednak nie uwzględniono tam charakteru porządkowego. Proszę to poprawić.

Zadania dodatkowe

Wykorzystać pakiet `dplyr`. Chętni dodatkowo mogą wypróbować alternatywne podejścia opisane w tym rozdziale. Oryginalny zbiór danych (około 200 tys. ofert) pochodzi z serwisu ogłoszeniowego *otomoto.pl* i został udostępniony przez *Fundację Naukową SmarterPoland.pl*. Do realizacji poniższych zadań dane poddałem obróbce, by ostatecznie liczbę ofert ograniczyć do 41 034.

Zad. 8. Wczytać plik z danymi `AutoSprzedam.dat` zapisując go w obiekcie o nazwie `auto`. Jeśli nazwa pierwszej kolumny jest zniekształcona użyć: `names(auto)[1] <- "NrOferty"`. Poprawność procedury można sprawdzić wykonując poniższe kroki (jeśli `FALSE` to źle).

```
> df_auto_spr <- readRDS("dane/df_auto_spr.rds") #Wczytujemy wzorzec
> identical(auto, df_auto_spr) #czy jest identyczny z wczytanymi danymi

[1] TRUE
```

Zad. 9. Stworzyć ramkę danych `df_niePolska`, w której oferty nie będą uwzględniały Polski jako kraju pochodzenia samochodu. Sprawdzenie według kroków:

```
> df_niePolska_spr <- readRDS("dane/df_niePolska_spr.rds") #Wczytujemy wzorzec
> identical(df_niePolska, df_niePolska_spr) #czy jest identyczny z wczytanymi danymi

[1] TRUE
```

Zad. 10. Stworzyć ramkę danych o nazwie `df_kraje3`, w której oferty będą uwzględniały tylko 3 najczęściej występujące kraje pochodzenia (pamiętać o usunięciu nieużywanych poziomów czynnika `droplevels`); powstałą ramkę zapisać do pliku i otworzyć w arkuszu kalkulacyjnym, pobieżnie sprawdzając poprawność eksportu. Aby sprawdzić, czy otrzymaliśmy właściwą ramkę danych wykonać poniższe kroki:

```
> df_kraje3_spr <- readRDS("dane/df_kraje3_spr.rds")
> identical(df_kraje3, df_kraje3_spr)

[1] TRUE
```

Zad. 11. Utworzyć ramkę danych (`df_kolor`), która zawiera tylko samochody w kolorze `czarny-metallic`. Kolumnę odnoszącą się do koloru samochodu usunąć. Sprawdzić według kroków:

```
>  df_kolor_spr <- readRDS("dane/df_kolor_spr.rds")
>  identical(df_kolor, df_kolor_spr)

[1] TRUE
```

Zad. 12. Dodać do ramki danych `auto` zmienną, która będzie ceną sprzedaży w EUR, z dokładnością do jednego miejsca po przecinku. Kurs wymiany to 4.19 PLN/EUR. Dodana kolumna (wyświetlam pierwszych 6 wierszy) wygląda tak:

```
  CenaEUR
1  6658.7
2  6682.6
3  6085.9
4  7136.0
5  7112.2
6  5107.4
```

Zad. 13. Utworzyć ramkę danych `df_akcyza`, która będzie składała się z 4 zmiennych: `PojemnoscSkokowa`, `CenaPLN`, `Akcyza` oraz `CenaAkcyza`. Pierwsze 2 zmienne występują w oryginalnym zbiorze danych `auto`. Z kolei akcyzę (`Akcyza`) wyliczamy według następującego schematu: samochody o pojemności nie przekraczającej 2000 cm^3 są opodatkowane stawką 3.1%, powyżej tej pojemności obowiązuje stawka 18.6%. Ostatnia ze zmiennych `CenaAkcyza` jest sumą ceny i akcyzy. Wersja trudniejsza: akcyza dotyczy samochodów sprowadzanych z zagranicy, dlatego wynosi 0, jeśli `KrajPochodzenia` to Polska. Wyniki dla pierwszych 6 wierszy:

```
> head(df_akcyza) #wersja łatwiejsza

  PojemnoscSkokowa CenaPLN Akcyza CenaAkcyza
1             1900   27900  864.9    28764.9
2             2000   28000  868.0    28868.0
3             1781   25500  790.5    26290.5
4             1991   29900  926.9    30826.9
5             2946   29800 5542.8    35342.8
6             1800   21400  663.4    22063.4
```

```
> head(df_akcyza2) #wersja trudniejsza

  KrajPochodzenia PojemnoscSkokowa CenaPLN Akcyza CenaAkcyza
1          Niemcy             1900   27900  864.9    28764.9
2          Polska             2000   28000    0.0    28000.0
3          Polska             1781   25500    0.0    25500.0
4          Polska             1991   29900    0.0    29900.0
5         Francja             2946   29800 5542.8    35342.8
6          Niemcy             1800   21400  663.4    22063.4
```

Zad. 14. Ze zbioru danych `auto` usunąć te obserwacje, dla których `RodzajPaliwa` to: `hybryda` lub `napęd elektryczny`. Następnie zrekodować zmienną na dwa poziomy: benzyna i olej napędowy. Tworząc tabelę dla tej zmiennej powinniśmy otrzymać następujące liczebności:

```
> table(df_auto$RodzajPaliwa)


      benzyna olej napedowy
        13102         27896
```

Rozdział 5

Eksploracyjna analiza danych

W tym rozdziale będziemy posługiwali się danymi, które zostały zebrane w wyniku przeprowadzenia, przez pewien bank portugalski, kampanii marketingu bezpośredniego. Dane pochodzą z *UCI machine learning repository*. Poniższa linijka wczytuje dane i zapisuje je w obiekcie o nazwie `bank`.

```
> ## Wczytujemy dane
> bank <- read.csv2("dane/bankFull.csv")
```

Zamieszczam również tabelę z oryginalnym opisem zmiennych. Jego tłumaczeniem zajmę się bezpośrednio w tekście.

Zmienna	Opis
age	(numeric)
job	(categorical) type of job: "admin.","unknown","unemployed","management","housemaid","entrepreneur", ...
marital	(categorical) marital status: "married","divorced","single"; note: "divorced" means divorced or widowed)
education	(categorical) "unknown","secondary","primary","tertiary"
default	(binary) has credit in default? "yes","no"
balance	(numeric) average yearly balance, in euros
housing	(binary) has housing loan? "yes","no"
loan	(binary) has personal loan? "yes","no"
contact	(categorical) contact communication type: "unknown","telephone","cellular"
day	(numeric) last contact day of the month
month	(categorical) last contact month of year: "jan", "feb", "mar", ..., "nov", "dec"
duration	(numeric) last contact duration, in seconds
campaign	(numeric) number of contacts performed during this campaign and for this client, includes last contact)
pdays	(numeric) number of days that passed by after the client was last contacted from a previous campaign, 1 means client was not previously contacted)
previous	(numeric) number of contacts performed before this campaign and for this client
poutcome	(categorical) outcome of the previous marketing campaign: "unknown","other","failure","success"
y	(binary) has the client subscribed a term deposit? "yes","no"

5.1. Agregacja danych z wykorzystaniem statystyk opisowych

W procesie agregacji danych dość często wykorzystujemy następujące miary: struktury, tendencji centralnej, pozycyjne oraz rozproszenia. Podstawą miary struktury jest liczba wystąpień każdej kategorii zmiennej, którą możemy oszacować wykorzystując funkcję `table()`. Jeżeli interesują nas częstości względne (odsetki), wtedy użyjemy kolejnej funkcji `prop.table(x, margin=NULL)`, w której `x` to liczebności, natomiast `margin` odnosi się do wymiaru dla którego obliczane są odsetki (np. wartość 1 to wiersze, wartość 2 to kolumny). Do przykładu wykorzystamy już wczytane do **R** dane `bank` i zmienną edukacja (`education`).

```
> ## Częstości (zlicza wystąpienia)
> (mojaTab <- table(bank$education))


  primary secondary  tertiary   unknown
     6851     23202     13301      1857
```

```
> ## Odsetki na podstawie mojaTab
> prop.table(mojaTab)


  primary secondary  tertiary   unknown
   0.1515    0.5132    0.2942    0.0411
```

Możemy również budować tabele dwu- i wielowymiarowe. Zobaczmy jak rozkładają się liczebności w dwuwymiarowej tabeli, skonstruowanej z uwzględnieniem zmiennej edukacja (`education`) oraz zmiennej mówiącej o nieregulowaniu należności (`default`).

```
> edu_def <- table(bank$default, bank$education)
> edu_def


     primary secondary tertiary unknown
  no    6724     22744    13103    1825
  yes    127       458      198      32
```

W zależności od celu, możemy policzyć odsetki dla wierszy (w każdym wierszu wartości sumują się do 1) lub dla kolumn (w każdej kolumnie wartości sumują się do 1):

```
> ## Odsetki dla wierszy, bo margin=1
> prop.table(edu_def, 1)


     primary secondary tertiary unknown
  no  0.1515    0.5123   0.2951  0.0411
  yes 0.1558    0.5620   0.2429  0.0393
```

```
> ## Odsetki dla kolumn, bo margin=2
> prop.table(edu_def, 2)


     primary secondary tertiary unknown
  no  0.9815    0.9803   0.9851  0.9828
  yes 0.0185    0.0197   0.0149  0.0172
```

Pozostałe miary zawiera poniższa tabela. Przypominam, że podane poniżej domyślne wartości argumentów zostaną uwzględnione przy obliczeniach, jeżeli całkowicie pominiemy je w funkcjach, np. pisząc `mean(x)`, **R** obliczy średnią z uwzględnieniem wartości pozostałych parametrów widocznych w poniższej tabeli.

Funkcja	Opis
`mean(x, trim = 0, na.rm = FALSE)`	Średnia lub średnia ucięta (`trim` - odsetek pominiętych) z wektora `x`; gdy `na.rm = TRUE`, wtedy brakujące dane są pomijane w obliczeniach
`median(x, na.rm = FALSE)`	Mediana z wektora `x`
`var(x, na.rm = FALSE)`	Wariancja z wektora `x`
`sd(x, na.rm = FALSE)`	Odchylenie standardowe z wektora `x`
`quantile(x, probs=seq(0, 1, 0.25), na.rm = FALSE)`	Kwantyle rzędu: 0, 0.25, 0.5, 0.75, 1
`IQR(x, na.rm = FALSE)`	Rozstęp międzykwartylowy, czyli `quantile(x, 0.75)-quantile(x, 0.25)`
`summary(x)`	Podaje wartości: największą, najmniejszą, średnią, medianę, pierwszy i trzeci kwartyl, liczbę braków danych

W zastawieniu brakuje jeszcze dwóch miar, które bywają pomocne w ocenie własności rozkładu, tj. miary asymetrii (skośności) i miary spłaszczenia (kurtozy). Nie ma ich w pakietach podstawowych, ale wykorzystując ich definicję (zob. wzory na str. 37) łatwo napisać odpowiednie funkcje. Poniżej je zamieszczam:

```
## Funkcja oblicza współczynnik asymetrii
asymetria <- function(x, na.rm=FALSE) {
   if (!is.numeric(x) || length(x) < 4){
       warning("argument nie jest numeryczny lub liczba elementow < 4: zwraca NA")
       return(NA_real_)
   }
   if (na.rm)
       x <- x[!is.na(x)] #usuń braki
   if (anyNA(x))
           return(NA_real_)

   n <- length(x)
   n/((n-1)*(n-2))*sum((x-mean(x))^3)/sd(x)^3
}

## Funkcja oblicza współczynnik spłaszczenia
kurtoza <- function(x, na.rm=FALSE) {
   if (!is.numeric(x) || length(x) < 4){
       warning("argument nie jest numeryczny lub liczba elementow < 4: zwraca NA")
       return(NA_real_)
   }
   if (na.rm)
       x <- x[!is.na(x)] #usuń braki
   if (anyNA(x))
           return(NA_real_)

   n <- length(x)
   n*(n+1)/((n-1)*(n-2)*(n-3))*sum((x-mean(x))^4)/sd(x)^4 - 3*(n-1)^2/((n-2)*(n-3))
}
```

Argument `na.rm` występuje we wszystkich funkcjach i należy o nim pamiętać, gdy występują braki danych. Poniższy przykład pokazuje zachowanie **R**, w dwóch sytuacjach.

```
> ## domyślnie na.rm=FALSE
> x <- c(1,5,3,NA,4,NA,7)
> mean(x)

[1] NA
```

```
> ## zmieniamy na na.rm=TRUE
> ## każąc wyrzucić barki z obliczeń
> mean(x, na.rm=TRUE)

[1] 4
```

Zanim przyjdziemy do bardziej rozbudowanych przykładów obliczmy kwantyle rzędu: 0.25, 0.30, 0.45, 0.95 dla zmiennej saldo rachunku (`balance`).

```
> ##  Zbiór danych: bank; percentyle: 25, 30, 45,  i 95 dla zm. balance
> quantile(bank$balance, probs=c(0.25, 0.3, 0.45, 0.95))

 25%  30%  45%  95%
  72  131  352 5768
```

Analiza danych z pakietem `dplyr`

Z pakietem `dplyr` zetknęliśmy się już w rozdziałach 4.2 i 4.3. Teraz wprowadzimy kolejne funkcje, zaczynając od `summarize()`[1]. Pozwala ona na agregację danych z wykorzystaniem funkcji napisanych przez użytkownika lub funkcji (statystyk) dostępnych w **R** (zob. tab. str. 53). Dodatkowo w ramach pakietu dostępne są:

- `n()` — zlicza wystąpienia kategorii zmiennej;

[1]Dopuszczalna jest również brytyjska pisownia funkcji wchodzących w skład pakietu.

- `n_distinct(x)` — podaje liczbę unikalnych wartości `x`;
- `first(x), last(x), nth(x, n)` — podaje dla zmiennej `x` obserwacje: pierwszą, ostatnią i n-tą.

Zanim przejdziemy do przykładu przypomnijmy zachowanie operatora %>% mówionego na str 46: zapis `x %>% f(a,b)` jest równoważny `f(x,a,b)`. Warto sie do niego przekonać, gdyż czytelność kody wzrasta. W kolejnych przykładach będę prezentował dwa warianty zapisu, by pokazać różnice. Później z tego zrezygnuję i będę używał operatora %>%.

Zadanie polega na wyliczeniu wartości statystyk tj. średniej, mediany, asymetrii i spłaszczenia dla zmiennej saldo rachunku (`balance`).

```
> ## Z wykorzystaniem operatora %>%
> bank %>%
+   summarize(srednia = mean(balance),
+     mediana = median(balance),
+     wspAsymetrii = asymetria(balance),
+     wspSplaszcz = kurtoza(balance))

  srednia mediana wspAsymetrii wspSplaszcz
1    1362     448          8.36       140.8
```

```
> ## Bez operatora %>%
> summarize(bank,
+     srednia = mean(balance),
+     mediana = median(balance),
+     wspAsymetrii = asymetria(balance),
+     wspSplaszcz = kurtoza(balance))

  srednia mediana wspAsymetrii wspSplaszcz
1    1362     448          8.36       140.8
```

Jeśli nie używamy operatora %>%, wtedy na pierwszej pozycji musi znaleźć się nazwa zbioru danych. W dalszej kolejności pojawiają sie interesujące nas funkcje, poprzedzone wymyślonymi nazwami kolumn. Sama funkcja nie byłaby specjalnie użyteczna, gdyby jej wywołanie ograniczało się do tak prostych zadań jak w powyższym przykładzie. Jej elastyczność docenimy w momencie, w którym użyjemy funkcji `group_by()` z pakietu `dplyr`. Funkcja ta tworzy grupy składające się z poziomów zmiennych nominalnych lub porządkowych. Jeżeli jej argumentem będzie nazwa zmiennej odnoszącej się do edukacji (`education`), wtedy **R** zapamięta, że funkcję `summarize()` należy zastosować do każdego poziomu tej zmiennej, czyli do każdego poziomu wykształcenia. Jeżeli nie korzystamy z operatora %>% wtedy pierwszym argumentem `group_by()` jest nazwa zbioru danych. Zobaczmy jak wyglądają statystyki dla salda rachunku w zależności od poziomu edukacji:

```
> ## Z wykorzystaniem operatora %>%
> bank %>%
+   group_by(education) %>%
+   summarize(srednia = mean(balance),
+     mediana = median(balance),
+     wspAsymetrii = asymetria(balance))

Source: local data frame [4 x 4]

  education srednia mediana wspAsymetrii
1   primary    1251     403        8.850
2 secondary    1155     392        8.535
3  tertiary    1758     577        7.431
4   unknown    1527     568        7.745
```

```
> ## Bez operatora %>%
> krok1 <- group_by(bank, education)
> summarize(krok1,
+     srednia = mean(balance),
+     mediana = median(balance),
+     wspAsymetrii = asymetria(balance))

Source: local data frame [4 x 4]

  education srednia mediana wspAsymetrii
1   primary    1251     403        8.850
2 secondary    1155     392        8.535
3  tertiary    1758     577        7.431
4   unknown    1527     568        7.745
```

Omówione dwie funkcje w połączeniu z już poznanymi funkcjami pakietu `dplyr` pozwalają w łatwy i bardzo sprawny sposób na manipulację danymi. Dla przypomnienia, a zarazem podsumowania, podaję ich krótką definicję:

- `select()` — pozwala wybrać zmienne (kolumny);
- `filter()` — pozwala wybrać przypadki (wiersze);
- `mutate()` — dodaje nowe kolumny;
- `arrange()` — sortuje wiersze;
- `summarize()` — pozwala wykorzystać statystyki podsumowujące (opisowe);
- `group_by()` — dzieli zbiór danych na grupy (ze względu na zmienne).

W tym przykładzie celem jest zbudowanie tabeli dla dwóch zmiennych: edukacja (`education`) i zgoda na lokatę terminową (`y`), w której znajdą się liczebności, częstości oraz procenty. Kolejność działań jest następująca: (a) podzielić zbiór danych ze względu na wspomniane zmienne dyskretne (b) zliczyć wystąpienia każdej kombinacji poziomów (funkcja `summarize()` oraz `n()`) co sprowadza się do podania liczebności (c) na podstawie dostępnych liczebności z poprzedniego punktu policzyć częstości oraz procenty (funkcja `mutate()`). Powyższe kroki w zapisie wyglądają tak:

```
> ## Z wykorzystaniem operatora %>%
> bank %>%
+   group_by(education, y) %>%
+   summarize(Liczeb = n()) %>%
+   mutate(Czestosc = Liczeb/sum(Liczeb),
+          Procent = 100*Czestosc)

Source: local data frame [8 x 5]
Groups: education

  education   y Liczeb Czestosc Procent
1   primary  no   6260    0.914    91.4
2   primary yes    591    0.086     8.6
3 secondary  no  20752    0.894    89.4
4 secondary yes   2450    0.106    10.6
5  tertiary  no  11305    0.850    85.0
6  tertiary yes   1996    0.150    15.0
7   unknown  no   1605    0.864    86.4
8   unknown yes    252    0.136    13.6
```

```
> ## Bez operatora %>%
> krok1 <- group_by(bank, y, education)
> krok2 <- summarize(krok1, Liczeb = n())
> mutate(krok2,
+     Czestosc = Liczeb/sum(Liczeb),
+     Procent = 100*Czestosc)

Source: local data frame [8 x 5]
Groups: y

    y education Liczeb Czestosc Procent
1  no   primary   6260    0.157    15.7
2  no secondary  20752    0.520    52.0
3  no  tertiary  11305    0.283    28.3
4  no   unknown   1605    0.040     4.0
5 yes   primary    591    0.112    11.2
6 yes secondary   2450    0.463    46.3
7 yes  tertiary   1996    0.377    37.7
8 yes   unknown    252    0.048     4.8
```

Uważny Czytelnik zapewne zastanawia się, dlaczego częstości i procenty różnią się między tabelami. Odpowiedź związana jest z kolejnością, w której kazaliśmy podzielić dane. W tabeli po lewej stronie najpierw pogrupowaliśmy ze względu na zmienną `education`, później ze względu na zmienną `y`. Jeżeli w kolejnym kroku użyjemy funkcji agregującej (a suma jest taką funkcją), wtedy obliczenia wykonywane są w obrębie wszystkich poziomów ostatniej zmiennej (tutaj `y`). Dlatego procenty sumują się do 100 w każdym `yes,no`. Odwrotnie jest dla tabeli po prawej stronie, tam procenty sumują się do 100 w obrębie poziomów edukacji.

W ostatnim przykładzie zobaczymy, jak przy użyciu poznanych funkcji możemy budować scenariusze analizy. Złóżmy, że interesują nas te obserwacje, dla których wynik ostatniej kampanii marketingowej (`poutcome`) zakończył się sukcesem lub porażką (pozostałe poziomy wyrzucamy). Wszystkie obliczenia wykonajmy w podziale na zmienne: edukacja (`education`), wynik ostatniej kampanii marketingowej (`poutcome`) oraz zgodna na lokatę terminową (`y`). Obliczmy liczebność i procenty oraz dla zmiennej saldo na rachunku (`balance`) średnią i medianę. W interpretacji i porównaniu otrzymanych statystyk chcemy ograniczyć się tylko do tych wartości, którym odpowiada zgoda na lokatę terminową — dlatego pozostałe należy usunąć. Dla ułatwienia porównania posortujmy kolumnę z wartościami procentowymi. W ostatnim kroku zawęźmy wyniki do wartości większych od 50 dla kolumny `Procent`. W analizie tego przykładu zachęcam do uruchamiania poniższego kodu partiami i obserwowania, jak zmieniają się wyniki w zależności od dodawania kolejnych linijek.

```
> bank %>%
+   filter(poutcome=="failure" | poutcome == "success") %>% #tylko sukces lub porażka
+   group_by(education, poutcome, y) %>%
+   summarize(sredniaSaldo = mean(balance),
+             medianaSaldo = median(balance),
+             Liczebnosc = n()) %>%
+   mutate(Procent=100*Liczebnosc/sum(Liczebnosc)) %>%
+   filter(y == "yes") %>% #weź tylko yes
+   select(-y) %>% #skoro samo yes, to wyrzuć zmienną y
+   arrange(desc(Procent)) %>%#sortuj malejąco
+   filter(Procent > 50)
```

```
Source: local data frame [4 x 6]
Groups: education, poutcome

  education poutcome sredniaSaldo medianaSaldo Liczebnosc Procent
1   unknown  success         2211          973         55    67.9
2  tertiary  success         2302          925        409    65.8
3 secondary  success         1610          703        433    64.1
4   primary  success         2487         1388         81    60.9
```

Ja zinterpretować wyniki? Krótki przykład zachęcający do samodzielnych wniosków: wśród osób o nieznanej edukacji (`education`), którzy na wcześniejszą kampanię marketingową odpowiedzieli pozytywnie (`poutcome`) prawie 68% wyraziło zgodę na utworzenie lokaty terminowej (`y`). Czyżby nadwyżka gotówki na koncie (`balance`) w porównaniu z innymi? Wynik należy interpretować z pewną ostrożnością, gdyż liczebność tej grupy jest niewielka.

Gdy w zbiorze danych jest bardzo dużo zmiennych, wtedy przydatna staje się szersza znajomość funkcji `select`. Poniższe sytuacje pokazują różne sposoby wyboru zmiennych.

```
select(mojeDane, plec, wiek) # tylko dwie zmienne
select(mojeDane, plec:wiek) # od plci do wieku (w kolejnosci)
select(mojeDane, contains("pyt")) # nazwy zawierajace pyt
select(mojeDane, starts_with("pyt_")) # zmienne zaczynajace sie od pyt_
select(mojeDane, ends_with("_r2014")) # zmienne konczace sie na _r2014
select(mojeDane, num_range("pyt", 1:4)) # od pyt1 do pyt4 (bez kolejnosci)
select(mojeDane, matches("...")) # pasujace do wyrazenia regularnego ...
```

5.2. Wizualizacja danych — system tradycyjny i pakiet graphics

Możliwości **R** w zakresie wizualizacji danych są ogromne. Generowanie grafiki odbywa się poprzez dwa interfejsy niskopoziomowe: system tradycyjny `graphics` oraz system `grid`. W ramach obu dostępnych jest bardzo wiele pakietów wspomagających i ułatwiających proces tworzenia grafiki. W tym rozdziale omówimy podstawowe funkcje z systemu tradycyjnego, by w następnym przejść do pakietu `ggplot2`. Pakiet ten został zbudowany z wykorzystaniem systemu `grid` i relatywnie niewielkim nakładem pracy pozwala tworzyć bardzo ładne wykresy[2].

Funkcje graficzne można podzielić na wysokopoziomowe i niskopoziomowe. Zadaniem tych pierwszych jest stworzenie kompletnego wykresu. Jeżeli ponownie funkcja z tej kategorii zostanie wywołana, wtedy tworzona jest nowa strona wykresu, a poprzedni zostaje zastąpiony, np. tak działa `plot()`[3]. Z kolei funkcje niskopoziomowe dodają kolejne elementy do już istniejącego wykresu i nie mogą być wywołane przed tymi pierwszymi. Przykładami takich funkcji są: `lines()`, `points()`, `abline()`.

Pamiętamy z przykładu (rysowanie gęstości) na str. 29, że najpierw wywołaliśmy funkcję `plot()` do narysowani jednej gęstości, a później posłużyliśmy się funkcją niskopoziomową `lines()` do narysowania kolejnych. Nie mogliśmy sekwencyjnie uruchamiać funkcji `plot()`, gdyż każdy poprzedni efekt zostałby zastąpiony aktualnym. Stąd potrzeba funkcji niskopoziomowych.

Pierwsza grupa omawianych funkcji pozwala rysować punty lub/i linie:

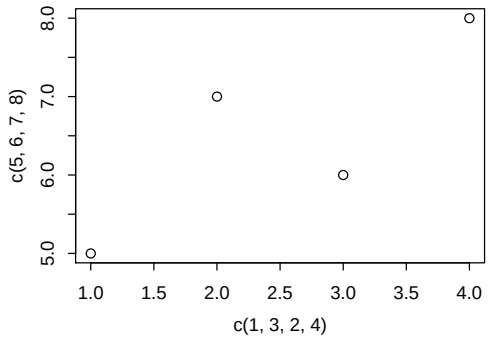
```
plot(x, y, ...)
lines(x, y, ...)
points(x, y, ...)
abline(a = NULL, b = NULL, h = NULL, v = NULL, ...)
```

[2] W ramach `grid` powstał też, dużo wcześniej, pakiet `lattice`.

[3] Duże możliwości **R** to również wiele odstępstw. Przykładowo, jeśli napiszemy `par(mfrow=c(2,3))` przed użyciem funkcji `plot()`, wtedy strona dzielona jest na 6 części (dwa wiersze i trzy kolumny). W konsekwencji możemy uruchomić 6 razy funkcję `plot()` otrzymując 6 różnych wykresów. Innym przykładem jest wykorzystanie (nie polecam): `par(new=TRUE)`.

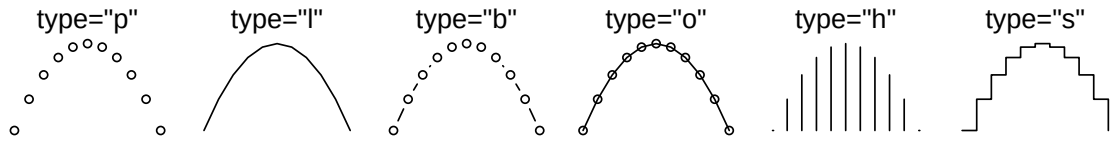
Argumentami funkcji są współrzędne osi X i Y. Gdy chcemy narysować zbiór punktów, wtedy `x` oraz `y` są wektorami o tej samej długości. Ilustruje to poniższy przykład:

```
> plot(c(1,3,2,4), c(5,6,7,8))
```

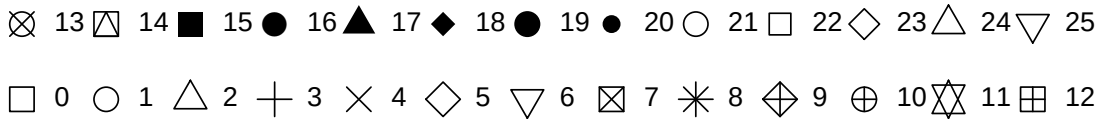


Lista pozostałych argumentów, które można wstawić zamiast kropek jest długa, o czym możemy się przekonać uruchamiając pomoc **R** w zakresie interesującej nas funkcji. Dodatkowo można zapoznać się z rozbudowanym zbiorem parametrów wpisując `?par` w konsoli. Ich mnogość może przytłaczać, dlatego wybrałem te absolutnie kluczowe, omawiając je poniżej. Przykłady będą ilustracją ich wykorzystania.

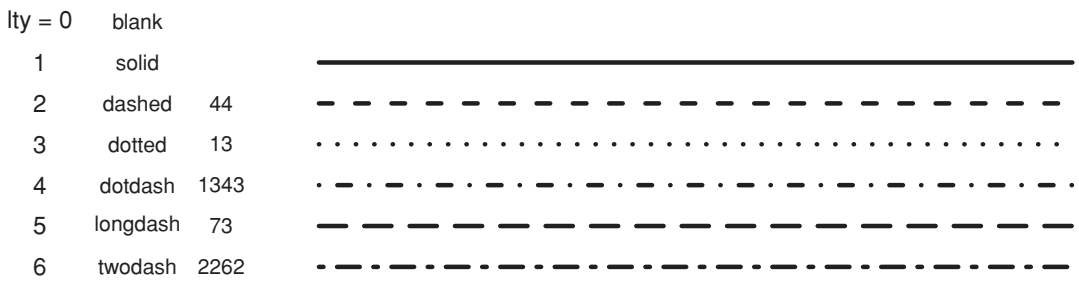
- `type` — pozwala kontrolować sposób prezentacji danych na wykresie; można określić, czy będą to punkty `"p"` (domyślne ustawienie), linia ciągła `"l"`, linia pozioma `"h"` itd. co widać poniżej:



- `pch` — symbol punktu podany jako liczba z zakresu od 0 do 25. Dodatkowo symbole od 21 do 25 mogą być wypełniane kolorem, jeśli dodatkowo użyjemy parametru `bg`, np. `bg="red"`.



- `lty` — parametr kontrolujący wzór linii. Podajemy wartości od 0 do 6 lub angielskie nazwy, np.: `lty="dotted"`:



- `lwd` — wartość tego parametru określa szerokość linii, domyślnie jest `lwd=1`. Teoretycznie, może to być każda liczba nieujemna.
- `col` — jest parametrem, którego wartościami może być nazwa koloru lub wektor składowych RGB. Dostępne nazwy można wyświetlić wpisując `colors()` w konsoli. Na pozycji 632 znajduje się kolor `tomato2`. Chcąc go użyć, powinniśmy napisać `col="tomato2"` lub równoważnie `col=rgb(238,92,66, maxColorValue=255)`. Składowe RGB tego koloru otrzymałem wpisując w konsoli `col2rgb("tomato2")`. Znając nazwę, raczej nie będziemy ze składowych korzystać.

```
> kolory <- colors() #zapisz nazwy 657 dostępnych kolorów
> sample(kolory, 10) #pokaz losowo wybranych 10 nazw

 [1] "wheat4"          "grey0"           "darkgoldenrod1" "cadetblue4"
 [5] "gray7"           "palegreen3"      "gray69"         "tomato2"
 [9] "darkslategray"   "grey37"
```

- `xlab, ylab, main` — opisy osi X i Y, tytułu wykresu, np. `main="To jest tytul wykresu"`.
- `xlim, ylim` — granice dla osi X i Y, np. `xlim=c(2, 10)` oznacza, że $x \in [2, 10]$.

Ciekawi kolorystyki mogą uruchomić poniższy skrypt, którego efektem są kolorowe prostokąty z liczbami w środku. Te wartości odpowiadają pozycji na której znajduje się dany kolor. Jeśli spodobał nam się kolor o numerze 641, wtedy wystarczy wpisać `colors()[641]` by poznać jego nazwę.

```
## Rysowanie palety kolorów od 1 do 650 (7 ostatnich pomijamy)
x <- seq(0, by=1.5, length.out = 25)
y <- seq(0, by=1, length.out = 26)
prost <- expand.grid(x, y)
prost <- cbind(prost, prost)
prost[,3] <- prost[,3] + 1.5
prost[,4] <- prost[,4] + 1
plot.new()
plot.window(c(0, max(x)+1.5), c(0, max(y)+1))
par(mar = rep(0,4), oma=rep(0,4))
kol <- rep("black", nrow(prost))
kol[c(153:200, 261:300)] <- "white"
for(wier in 1:nrow(prost)) {
   rect(prost[wier,1], prost[wier,2],prost[wier,3],prost[wier,4], col=colors()[wier])
   text(prost[wier,1]+0.75, prost[wier,2] + 0.5, wier, cex=0.6, col=kol[wier])
}
```

W którym miejscu zmienić powyższy skrypt, aby otrzymać nazwy kolorów zamiast numerów. Poniżej efekt — czytelność nazw możliwa, po dużym powiększeniu.

W dalszym ciągu będziemy wykorzystywali dane `bank`. Zakładam, że są one wczytane do pamięci **R**. Zazwyczaj dla łatwiejszego porównania i oszczędzenia miejsca, umieszczam wykresy obok siebie. Należy pamiętać, że pierwszemu użyciu funkcji wysokopoziomowej (np. `plot()`) odpowiada wykres po lewej stronie.

Poniżej przedstawiam podstawowe możliwości systemu tradycyjnego. Zdecydowanie więcej miejsca poświęcimy pakietowi `ggplot2`.

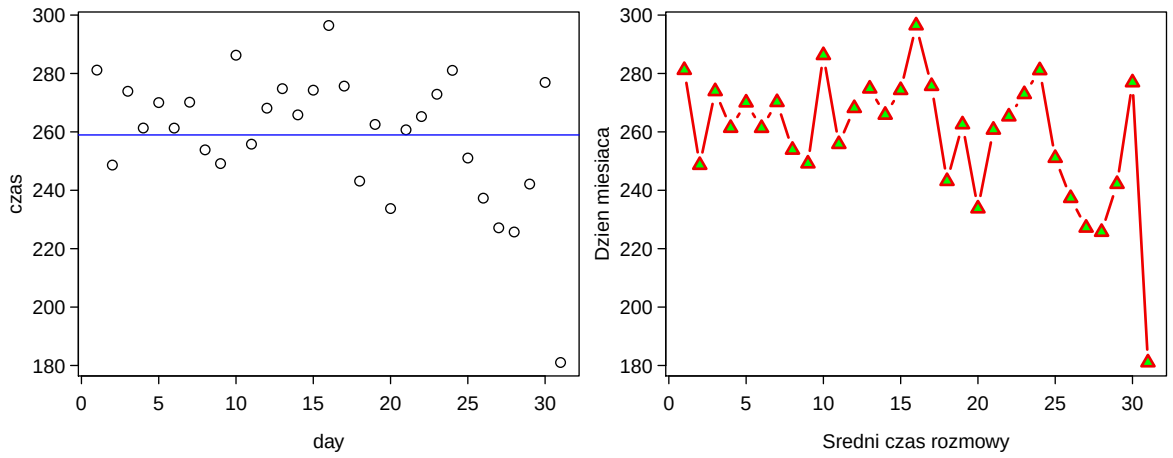
Wykres liniowy i punktowy

W tym przykładzie chcemy zobaczyć, jaki jest średni czas rozmowy (duration) w każdym dniu miesiąca (day). Zanim narysujemy wykres, musi taką informację wydobyć z danych, wykorzystując poznane do tej pory funkcje pakietu dplyr.

```
> ## Ramka danych: bank; tworzymy ramkę dayCzas; pamiętać o library(dplyr)
> dayCzas <- bank %>%
+    group_by(day) %>%
+    summarize(czas = mean(duration))
> head(dayCzas) #pokaż 6 pierwszych

  day czas
1   1  281
2   2  249
3   3  274
4   4  261
5   5  270
6   6  261
```

Tylko w poniższym przykładzie jawnie użyję par, by zwrócić uwagę na dodatkowe możliwości „upiększające" efekt końcowy. W nawiasach [] podaję wartości domyślne, które zostaną użyte, gdy ich nie zmienimy.

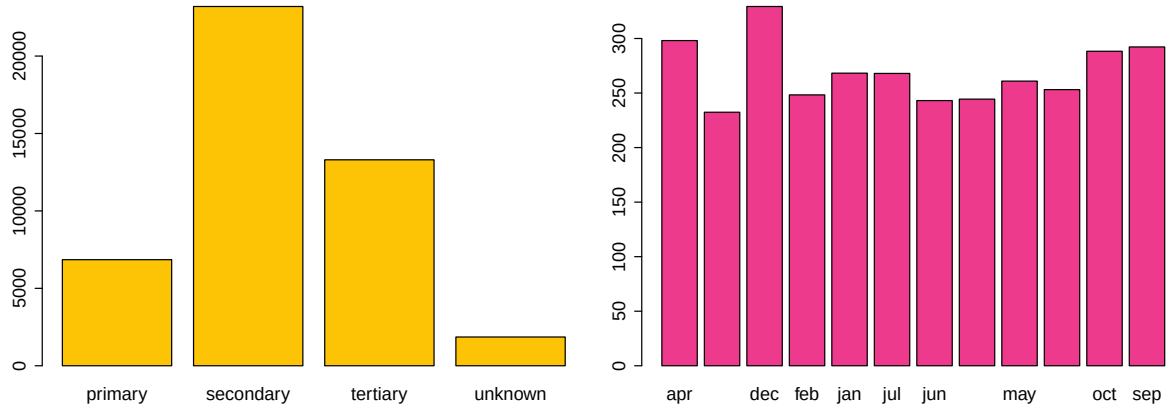
```
> par(mar = c(3, 3, 0.4, 0.1),#marginesy: dolny, lewy, górny, prawy [c(5.1,4.1,4.1,2.1)]
+     cex.lab = 0.95,  #skalowanie opisów osi [1]
+     cex.axis = 0.9,  #skalowanie osi [1]
+     mgp = c(2, 0.5, 0),  #w której linii: opis, etykieta, oś [c(3,1,0)]
+     tcl = -0.3,  #długość znaczników osi [-0.5]
+     las=1)  # 1 - poziomo,  0 - równolegle; wyświetlane wartości na osi y [0]
> plot(dayCzas) #Domyślne wszystkie ustawienia
> abline(h = mean(dayCzas$czas), col="blue") #dodaj horyzontalną linię = średniej
> plot(dayCzas, type="b", col="red2", lwd=2, pch=24, bg="green",
+ xlab="Sredni czas rozmowy", ylab="Dzien miesiaca") #wykres po prawej
```



Wykresy słupkowe

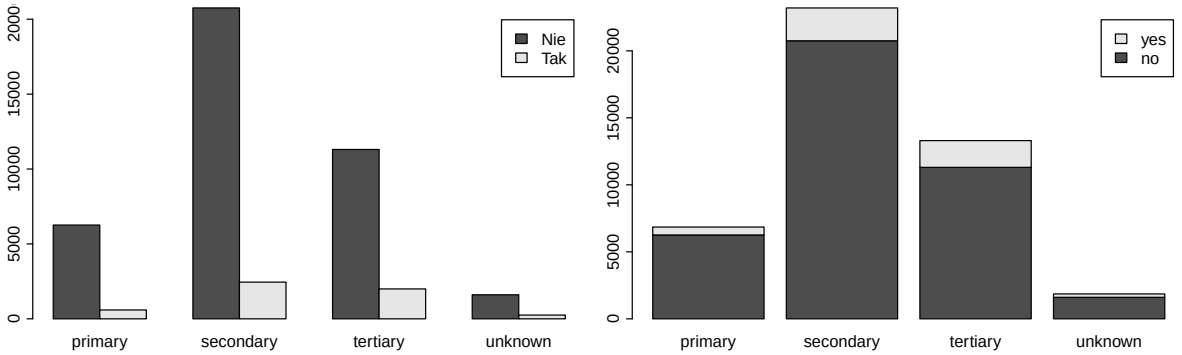
Funkcja barplot() pozwala przedstawić dane w postaci słupków. Argumentem wejściowym są wartości, odpowiadające wysokości słupka. Dlatego konieczne będzie wstępne przetworzenia danych. Jeśli chcemy na wykresie zobaczyć, jak kształtują sie liczebności poziomów edukacji (zmienna education), wtedy najlepiej użyć funkcji table(), a jej wynik wpisać jako argument wejściowy barplot() — tak powstał pierwszy wykres. Z kolei na drugim wykresie przedstawiony jest średni czas trwania rozmowy (duration) w każdym miesiącu. Tutaj też nastąpiło wstępne przetworzenie danych.

```
> monCzas <- bank %>%
+    group_by(month) %>%
+    summarize(czas = mean(duration))
> barplot(table(bank$education), col=rgb(253,196,6, maxColorValue = 255))
> barplot(monCzas$czas, col="violetred2", names=monCzas$month)
```



Zobaczmy, w jaki sposób dwuwymiarowa tabela jest odwzorowywana w wykres słupkowy z legendą. Ustawiony parametr `beside=TRUE` zapewni wyświetlenie słupków obok siebie.

```
> (tab <- table(bank$y, bank$education))


     primary secondary tertiary unknown
  no    6260     20752    11305    1605
  yes    591      2450     1996     252

> barplot(tab, legend=c("Nie", "Tak"), beside=TRUE) #słupki obok siebie
> barplot(tab, legend=TRUE, beside=FALSE) # beside=FALSE jest domyślne, można wyrzucić
```



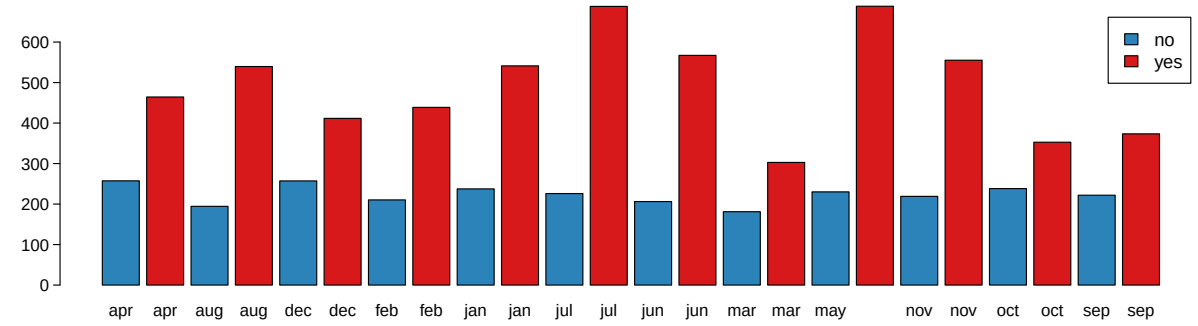
Teraz chcemy zobaczyć, jak rozkłada się średni czas trwania rozmowy (`duration`) w poszczególnych miesiącach (`month`) ze względu na wyrażenie zgody na lokatę (`y`). Najpierw trzeba przygotować dane.

```
> monYCzas <- bank %>%
+    group_by(month, y) %>%
+    summarize(czas = mean(duration))
>    head(monYCzas) #dane do przekształcenia

  month   y czas
1   apr  no  257
2   apr yes  464
3   aug  no  194
```

```
4   aug yes  540
5   dec  no  257
6   dec yes  412
```

To przekształcenie jest niewystarczające, gdyż trudno otrzymać efekt, który w pełni nas zadowoli. Bez dodatkowych „trików” otrzymamy wykres, w którym podpisy pod osią X będą sie powtarzały. Użyte poniżej argumenty funkcji są niezbędne, aby powstał wykres.

```
> barplot(monYCzas$czas, names.arg = monYCzas$month, legend.text = unique(monYCzas$y),
+         col=c("#2b83ba", "#d7191c"))
```



Najlepszym więc rozwiązaniem jest restrukturyzacja danych (zob. efekt poniżej) do dwuwymiarowej tabeli, w której każda kolumna odpowiadała kolejnemu miesiącowi, natomiast wiersze odpowiadają kategoriom zmiennej `y`. Funkcja `dcast()` z pakietu `reshape2` służy m.in. do takich celów.

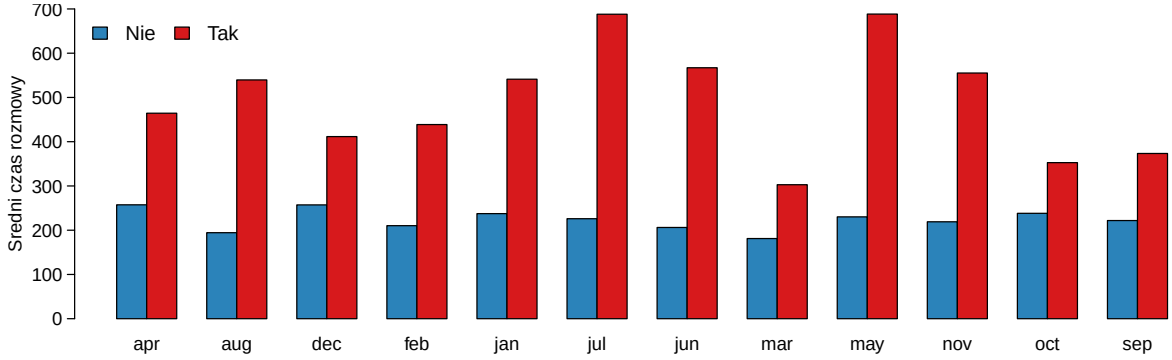
```
> library(reshape2) # aby restrukturyzować dane z monYCzas -> myc
> myc <- dcast(monYCzas, y ~ month)   #po przekształceniu
> myc # struktura odpowiednia do barplot

    y apr aug dec feb jan jul jun mar may nov oct sep
1  no 257 194 257 210 237 226 206 181 230 219 238 222
2 yes 464 540 412 439 541 688 567 303 688 555 353 373
```

Słupki przedstawiają wartości znajdujące się w kolumnach od 2 do 13 — to są dane wejściowe, które muszą być typu macierz. Za legendę odpowiada pierwsza kolumna. Choć wystarczyłby napisać:

```
barplot(as.matrix(myc[,2:13]), legend = myc$y, beside=TRUE)
```

to jednak rozszerzymy listę argumentów, pokazując dodatkowe możliwości. Jak się przekonamy później, użycie pakietu `ggplot2` nie wymaga takich zabiegów.

```
> barplot(as.matrix(myc[,2:13]), legend = myc$y, beside=TRUE,
+     col=c("#2b83ba", "#d7191c"), ylim=c(0, 700), ylab="Sredni czas rozmowy",
+     args.legend = list(x="topleft", legend=c("Nie", "Tak"), ncol=2, box.lty=0))
```



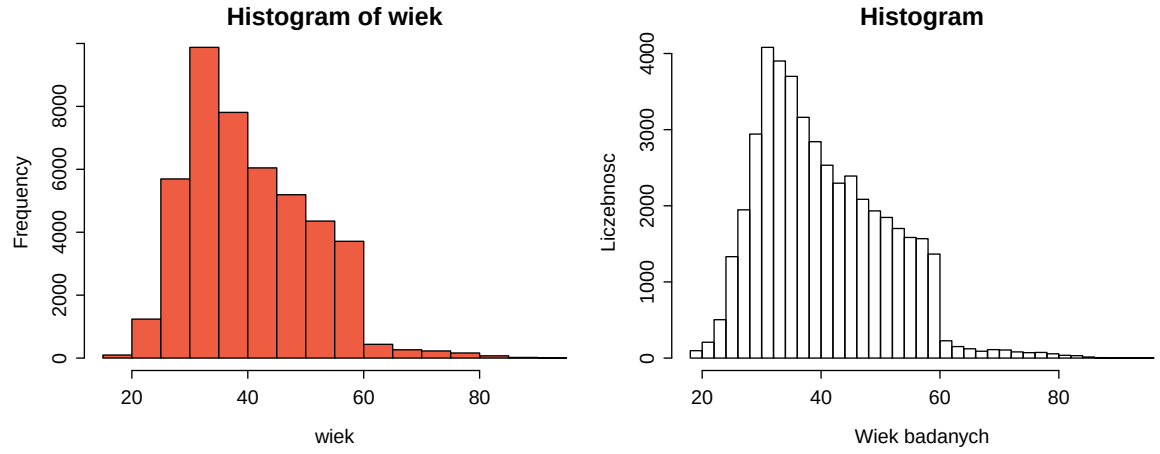
Inne pozycje legendy to: `"bottomright"`, `"bottom"`, `"bottomleft"`, `"left"`, `"topleft"`, `"top"`, `"topright"`, `"right"`, `"center"`.

Histogram, gęstość, dystrybuanta i pudełko-wąsy

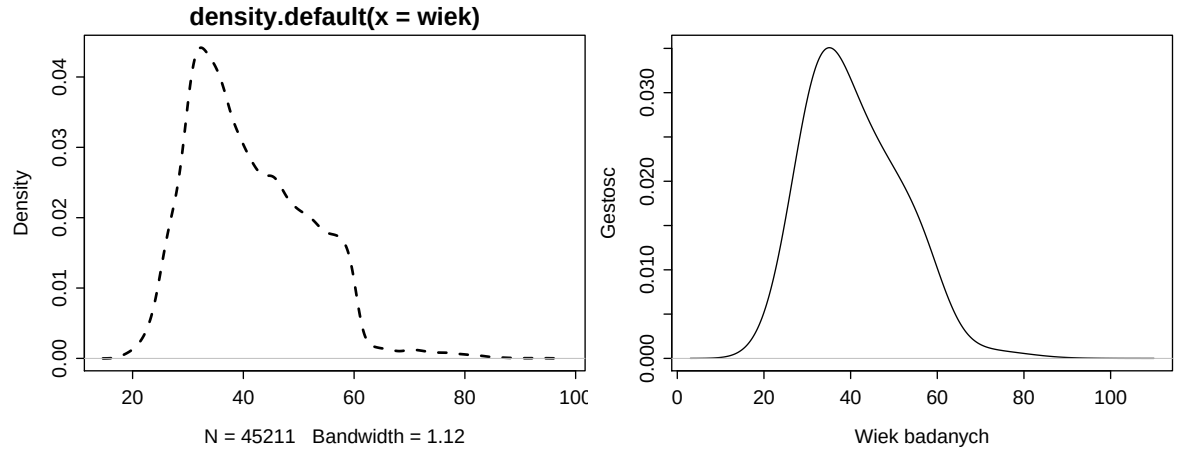
```
hist(x, breaks =..., main = ..., xlab =..., ylab=...,)
density(x, bw=..., lwd=..., lty=...)
ecdf(x)
boxplot(x)
```

W histogramie `breaks` odpowiada liczbie klas. Szerokość okna w gęstości to `bw`; im większa wartość, tym gęstość będzie bardziej wygładzona. Funkcje gęstości `density()` oraz dystrybuanty `ecdf()` dodatkowo wymagają wywołania funkcji `plot()`. Ich użycie ilustrują poniższe przykłady.

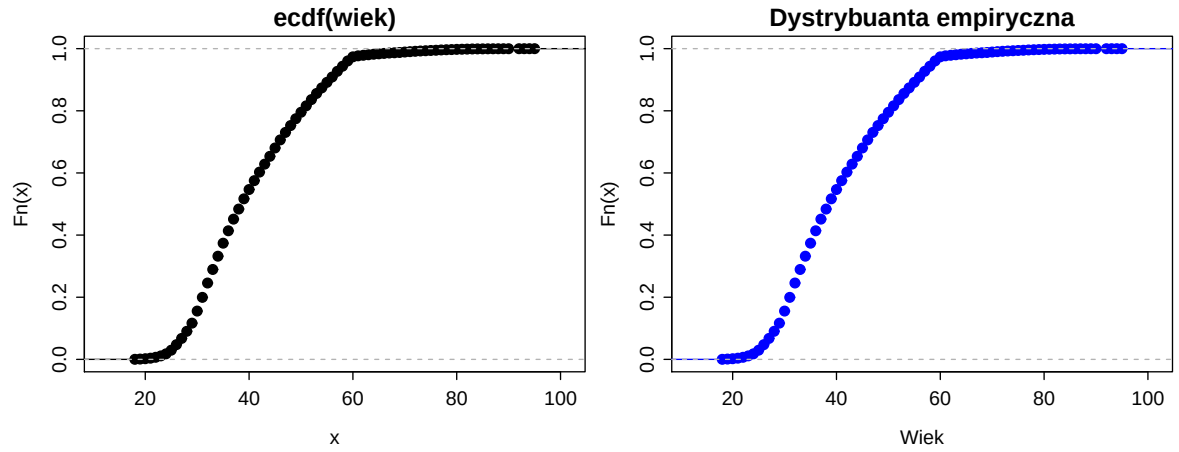
```
> ## Ramka danych: bank; wykresy dal zmiennej age
> wiek <- bank$age
> hist(wiek, col="tomato2") #domyślna liczba przedziałów i opisów
> hist(wiek, breaks = 30, main="Histogram", xlab="Wiek badanych", ylab="Liczebnosc")
```



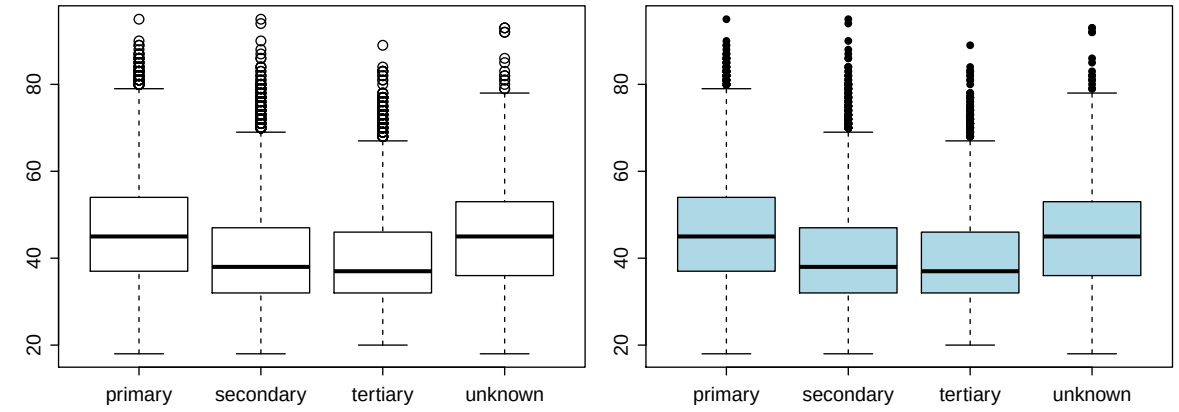

```
> ## Dane: wiek z poprzedniego przykładu
> plot(density(wiek), lwd=2, lty=2) #domyślny poziom wygładzenia bw (Bandwidth)
> plot(density(wiek, bw=5), main="", xlab="Wiek badanych", ylab="Gestosc")
```




```
> ## Dane: wiek z poprzedniego przykładu
> plot(ecdf(wiek))
> plot(ecdf(wiek), col="blue", main="Dystrybuanta empiryczna", xlab="Wiek")
```




```
> ## Ramka danych: bank
> boxplot(age ~ education, data=bank)
> boxplot(age ~ education, data=bank, col="lightblue", pch=20)
```



5.3. Grafika z pakietem ggplot2

Pakiet[4] jest implementacją gramatyki grafiki Wilkinsona — stąd w początkowej nazwie gg. Gramatyka to zbiór reguł mówiących o zasadach tworzenia grafiki. Odpowiada ona na pytanie: czym jest grafika statystyczna. Pierwszym elementem jest tzw. estetyka (*aesthetic*), która jest odwzorowaniem danych w atrybuty wykresu takie jak: kolor, kształt, rozmiar. Mówi nam o roli jaką każda zmienna pełni na wykresie. Przykładowo, jedna zmienna będzie odwzorowywana na osi X, druga na osi Y, a trzecia będzie odwzorowana w postaci koloru (a może kształtu) i umieszczona w legendzie. Mając na względzie estetykę, pierwszym krokiem w tworzeniu wykresu jest podanie nazwy ramki danych oraz przyporządkowanie ról poszczególnym zmienny. W tym celu wykorzystujemy następującą funkcję, w której po znaku równości podajemy nazwy zmiennych (zwróć uwagę na skrót *aes* pochodzący od *aesthetic*):

```
ggplot(ramkaDanych, aes(x=..., y=..., color=..., fill=..., shape=...))
```

Oczywiście nie wszystkie elementy estetyki będą wykorzystywane jednocześnie. Jeśli chcemy narysować prosty wykres słupkowy pokazujący liczebności poszczególnych kategorii zmiennej dyskretnej o nazwie `edukacja`, wtedy wystarczy zdefiniować oś X, czyli wpisać `x=edukacja`.

Drugim elementem (lub warstwą) jest atrybut geometryczny mówiący o tym, w jaki sposób dane będą prezentowane na wykresie (zaczynają się od `geom_`). Mogą to być to punkty, słupki, linie itp. Ten element należy dodać do poprzedniego używając znaku +. W nawiązaniu do powyższego przykładu, powinniśmy

[4]Dokumentacja `ggplot2` jest również dostępna pod oficjalnym adresem: `http://docs.ggplot2.org/current/`

wykorzystać funkcję geom_bar(), aby narysować wykres słupkowy. Poniższy zapis pozwoli taki wykres wygenerować:

```
ggplot(ramkaDanych, aes(x=edukacja)) + geom_bar()
```

W niniejszym opracowaniu będziemy wykorzystywać następujące funkcje z tej grupy:

geom_line(), geom_point(), geom_smooth(), geom_bar(), geom_histogram(), geom_density(), geom_boxplot()

Obok elementów geometrycznych występują statystyczne (zaczynają się od stat_), których zadaniem jest przekształcanie zmiennych z wykorzystaniem elementów statystyki. Nas będą interesować następujące funkcje:

```
stat_function(), stat_ecdf(),
```

Kolejnym elementem są skale (zaczynają się od scale_). Pozwalają one kontrolować sposób, w jaki dane są odwzorowywane w elementy estetyczne. W tym miejscu możemy decydować o kolorystyce, kształcie obiektów, rozmiarze. To jest element opcjonalny (nie jest wymagany do narysowania wykresu), który zmieniamy wtedy, gdy nie jesteśmy zadowoleni z domyślnych ustawień. Używać będziemy następujących:

scale_fill_brewer(),scale_fill_manual(),scale_color_brewer(),scale_color_manual scale_size_manual(),scale_shape_manual(),scale_linetype_manual()

Współrzędne stanowią kolejny element (zaczynają się od coord_). Szczególnie przydatne będą funkcje, które pozwolą definiować przedziały dla osi X i Y, zamieniać osie (transponować) oraz kontrolować skalę czy proporcję między osiami:

```
coord_cartesian(), coord_fixed(),coord_flip()
```

Ostatni element (*facets*) odnosi się do przedstawiania wielu wykresów w tzw. panelach. Idea polega na tym, że dane są dzielone ze względu na poziomy zmiennej dyskretnej, i dla każdego poziomu tworzony jest osobny wykres. To zagadnienie zostało przeniesione do rozdz. 5.3, w którym poruszam bardziej zaawansowane kwestie pakietu.

Istotną cechę przedstawionej gramatyki jest możliwość dodawania kolejnych warstw. To bez wątpienia daje dużą elastyczność i pozwala tworzyć skomplikowane wykresy. Zanim przejdziemy do szczegółowych zagadnień, poświęćmy chwilę kolorystyce.

Do tej pory poznaliśmy dwa sposoby wymuszania kolorystyki (zob. str. 58): poprzez podanie nazwy koloru lub składowej RGB. Jeśli do tej pory się nad tym nie zastanawialiśmy, to w miarę zdobywania doświadczenia zadamy sobie następujące pytania: jak łączyć ze sobą kolory, które wybierać, aby na wydruku czarno-białym widoczne były różnice, które kolory są przyjazne dla osób mających problem z ich odróżnianiem, a wreszcie które nadają się do wydruków kolorowych. Odpowiedzią na te pytania jest pakiet RColorBrewer, a wszystko jest opisane i pięknie przedstawione na stronie http://colorbrewer2.org/. Można też w konsoli wpisać display.brewer.all(), aby wyświetli palety.

Pakiet ggplot2 wspiera użytkownika w możliwościach wyboru palet z RColorBrewer. Wszystkie scale_ kończące się na brewer zmieniają kolorystykę, według zdefiniowanej przez użytkownika palety. Wystarczy tylko podać jej nazwę. Przykładowo, jeśli chcemy użyć palety o nazwie Set1 i zmienić wypełnienie (*fill*) słupków (zmienna ma 3 poziomy i znajduje się w legendzie), to należy dodać do poprzednich warstw: scale_fill_brewer(palette="Set1"). Funkcja sama pobierze 3 najlepsze kolory.

Istnieje również możliwość ręcznego zdefiniowania kolorów i odbywa się to za pomocą funkcji kończących się na (_manual). Wtedy zobligowani jesteśmy do podania składowych RGB lub skorzystania z zapisu szesnastkowego (HEX). Na wspomnianej wcześniej stronie oprócz nazwy palety, pojawiają się też RGB i HEX. Używając tego ostatniego — i chcąc osiągnąć identyczny efekt do tego z użyciem palety Set1 — należałoby napisać: scale_fill_manual(values=c("#e41a1c", "#377eb8", "#4daf4a")).

Punkty i linie

Linie, punkty czy krzywe wygładzone naniesiemy na wykres wykorzystując poniższe funkcje.

```
geom_line(linetype=..., col=..., size=..., alpha=...)
geom_point(shape=..., col=..., fill=..., size=..., alpha=...)
geom_smooth(method=..., se=FALSE)
```

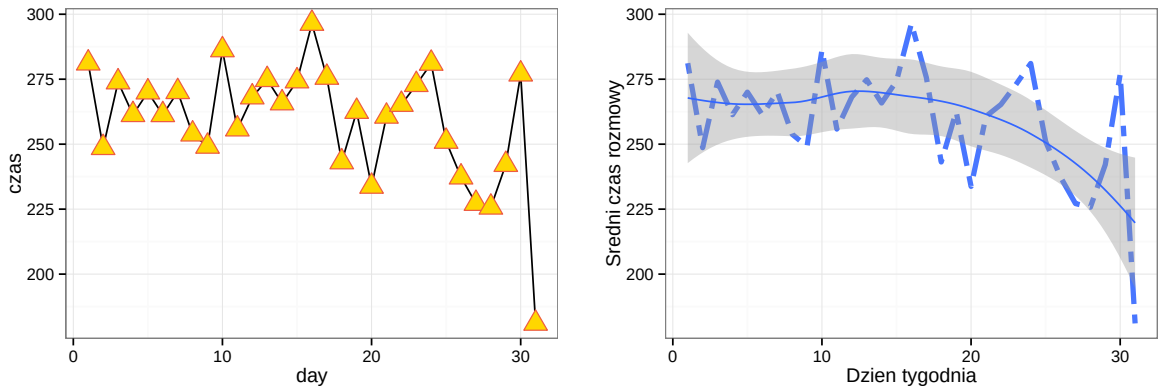
Wszystkie przedstawione wewnątrz argumenty mają charakter opcjonalny. Wzór linii (`linetype`) oraz kształt punktu (`shape`) odpowiadają wartościom parametrów wykorzystywanych w tradycyjnym systemie graficznym, czyli są odpowiednio synonimami dla `lty` oraz `pch` (zob. str. 58). Również `col` oraz `fill` są synonimami kolorów dla `col` i `bg`. Z kolei rozmiar (`size`) podawany jest w milimetrach. Nowością jest parametr `alpha`, który mówi o stopniu pokrycia kolorem. Wartość 1 oznacza 100% pokrycie, natomiast 0 całkowitą przezroczystość.

Jak w kilku miejscach podkreślono, wykresy tworzymy nakładając kolejne warstwy. Tym samym możemy najpierw narysować punkty, później je połączyć linią, a ostatecznie nanieść krzywą wygładzoną. W poniższym przykładzie zademonstrujemy wykorzystanie tych funkcji, a użyjemy do tego wcześniej stworzonej ramki danych `dayCzas`. Przypomnijmy, jak wyglądała:

```
>  head(dayCzas,3)

  day czas
1   1  281
2   2  249
3   3  274
```

Przedstawia ona średni czas rozmowy w każdym dniu miesiąca. W pierwszym kroku należy określić tzw. estetykę. Na osi X będzie zmienna `day`, a na osi Y znajdzie się `czas`. Później dodajemy kolejne warstwy. Jeśli warstwa lub warstwy zostaną zapisane w postaci obiektu, wtedy efekt (wykres) będzie widoczny po wpisaniu w konsoli nazwy tego obiektu — analogicznie do przypisania `x <- 1`. Gdy chcemy wyświetlić wartość, wtedy w konsoli wpisujemy `x`.
Jeszcze jedna uwaga: styl (*theme*) wykresu można modyfikować. Domyślny, w którym tło wykresu jest szare niezbyt przypadł mi do gustu, dlatego zawsze będę dodawał styl `theme_bw()`. Poniżej znajdują się dwa wykresy (pierwszy z lewej strony, drugi z prawej).

```
> library(ggplot2) #wczytujemy pakiet
> p1 <- ggplot(dayCzas, aes(x=day, y=czas)) + theme_bw() #do tego dodamy inne warstwy
> p1 + geom_line() + geom_point(shape=24, size=5, col="tomato2", fill="gold") #lewy wyk.
> p1 + geom_line(linetype=6, col="royalblue1", size=1.5) +
+      xlab("Dzien tygodnia") + ylab("Sredni czas rozmowy") + geom_smooth() #prawy wyk.
```

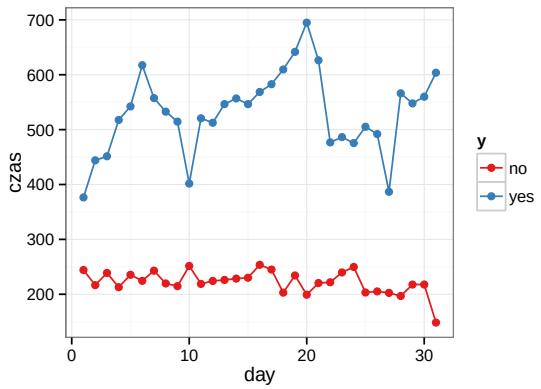


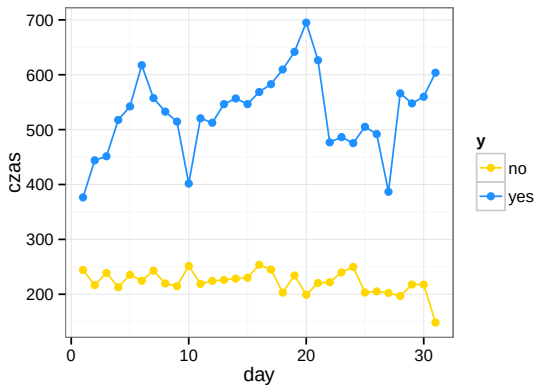
Jeśli rysujemy jedną linię lub jeden zbiór punktów, wtedy kolorystykę czy kształt wymuszamy odpowiednim argumentem w elemencie graficznym, jak pokazano powyżej. Gdy są dwie linie lub więcej, i każda ma mieć inny kolor, wtedy musimy użyć elementu `scale_`. Aby to pokazać, stwórzmy nową ramkę danych, w której uwzględnimy zmienną: zgoda na lokatę terminową `y`.

```
> dayCzas2 <- bank %>%
+    group_by(day, y) %>%
+    summarize(czas = mean(duration))
> head(dayCzas2, 4)

  day   y czas
1   1  no  244
2   1 yes  376
3   2  no  217
4   2 yes  444
```

Budujemy wykres podobny do poprzednich z tą różnicą, że na nim pojawią się dwie linie odpowiadające: `yes,no` zmiennej `y`. Pamiętać należy, że w tej sytuacji element estetyczny zostanie rozszerzony o tą dodatkową zmienną. Będzie to zmienna grupująca, która musi być typu czynnik (ta jest). Jeśli chcemy dokonać różnienia między kategoriami zmiennej `y` za pomocą koloru, wtedy należy napisać: `color=y`. Możemy dobrać też różne kształty dla punktów, wtedy zapiszemy `shape=y`. Niech nie zmyli nas nazwa tej zmiennej. Nie ma ona nic wspólnego z elementem estetycznym `y=...`; to zwykły zbieg okoliczności, a jeśli komuś to przeszkadza, nazwę tej zmiennej można zmienić. Wybierzmy kolor jako podstawę rozróżnienia kategorii.

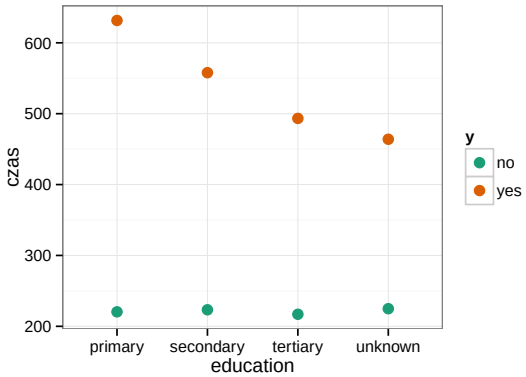
```
> p2 <- ggplot(dayCzas2, aes(x=day, y=czas, color=y)) + theme_bw()
> p2 <- p2 + geom_line() + geom_point(shape=19, size=2)
> p2 + scale_color_brewer(palette="Set1")                #sprawdź też komentarz niżej
> p2 + scale_color_manual(values=c("gold","dodgerblue")) #c(yes="gold",no="dodgerblue")
```





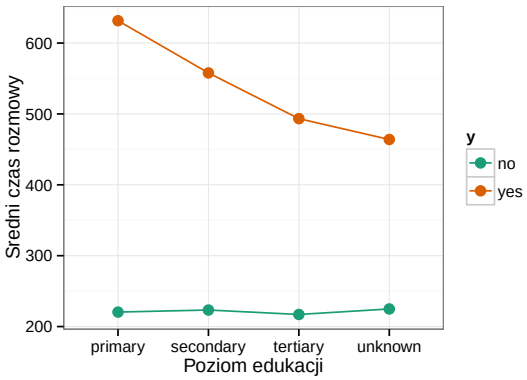
Kolejny przykład i kolejne wyzwanie. W oparciu o ramkę danych `eduCzas` zbudować wykres, w którym punkty będą ze sobą połączone (zobacz niżej). Ale które punkty **R** ma ze sobą połączyć, jaką zasadą ma się kierować?

```
> eduCzas <- bank %>%
+    group_by(education, y) %>%
+    summarize(czas = mean(duration))
> eduCzas

  education   y czas
1   primary  no  220
2   primary yes  632
3 secondary  no  223
4 secondary yes  558
5  tertiary  no  217
6  tertiary yes  493
7   unknown  no  225
8   unknown yes  464
```



Przeanalizujmy sytuację: na osi X mamy zmienną dyskretną edukacja. Linia jest przewidziana dla zmiennych ciągłych — tak np. były traktowane dni w poprzednim przykładzie i **R** wiedział, że ich połączenie jest prawidłowe. Dlatego trzeba powiedzieć **R**, które punkty ma ze sobą połączyć. Łączenie punktów w pionie dostarcza innych informacji niż łączenie ich w poziomie. Decyzja dotyczy łączenia w ramach kategorii `y` lub kategorii `education`, co implikuje wpisanie wewnątrz `aes` odpowiednio: `group =y` lub `group=education`. Wybieramy to pierwsze rozwiązanie, gdyż chcemy pokazać tendencję dla zmiennej porządkowej `education`.

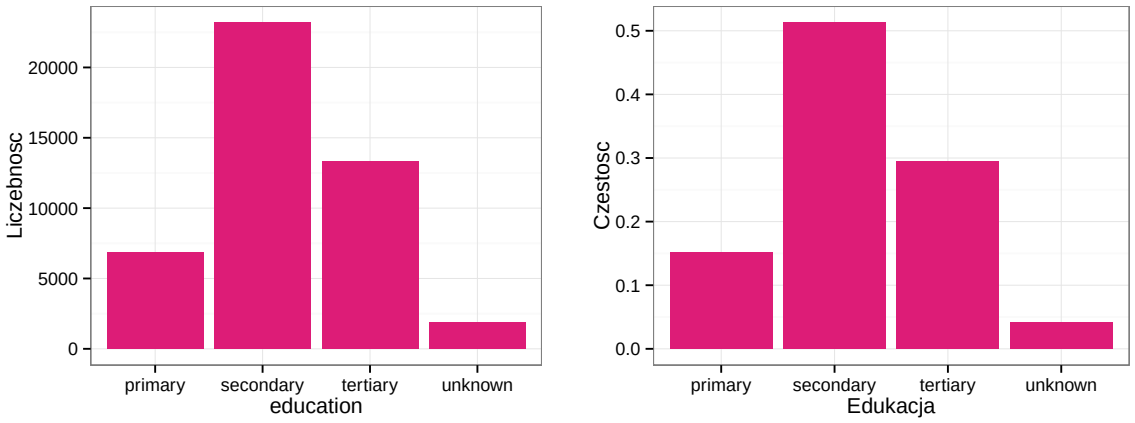
```
> p2 <- ggplot(eduCzas,
+              aes(x=education,
+                  y=czas,
+                  color=y,
+                  group=y)) + #tutaj zmiana
+       theme_bw() +
+       geom_line() +
+       geom_point(shape=19, size=3) +
+       scale_color_brewer(palette="Dark2")
> p2 + xlab("Poziom edukacji") +
+      ylab("Sredni czas rozmowy")
```



Wykresy słupkowe

Wykres słupkowy rysujemy nanosząc warstwę geometryczną za pomocą `geom_bar()`. Gdy mamy zmienną nominalną bądź porządkową i chcemy, aby wysokości słupka odpowiadała liczebności bądź częstości (procent) poszczególnych kategorii zmiennej, wtedy najszybszym podejściem jest wykorzystanie oryginalnych danych. Funkcja `geom_bar()` automatycznie zlicza wystąpienia, dlatego w `aes` podajemy tylko nazwę zmiennej dla pozycji `x`. Jeśli zamiast liczebności chcemy otrzymać częstości, wtedy trzeba dodatkowo w `aes` napisać: `y=(..count..)/sum(..count..)`. Można też po znaku = wpisać `100*`, by mieć procenty. Prześledźmy te dwa warianty rysowania na poniższym przykładzie.

```
> ## Wykresy słupkowe; oryginalne dane: bank
> ## Gdy chcemy wypełnić kolorem, wtedy należy podać kolor po fill=
> ## Oprócz nazwy koloru, składowych RGB można użyć systemu HEX
> p1 <- ggplot(bank, aes(x=education)) + theme_bw()
> p1 + geom_bar(fill="#dd1c77") + ylab("Liczebnosc") #wyk. strona lewa
> p2 <- ggplot(bank, aes(x=education, y=(..count..)/sum(..count..))) + theme_bw()
> p2 + geom_bar(fill="#dd1c77") + ylab("Czestosc") + xlab("Edukacja") #wyk. strona prawa
```



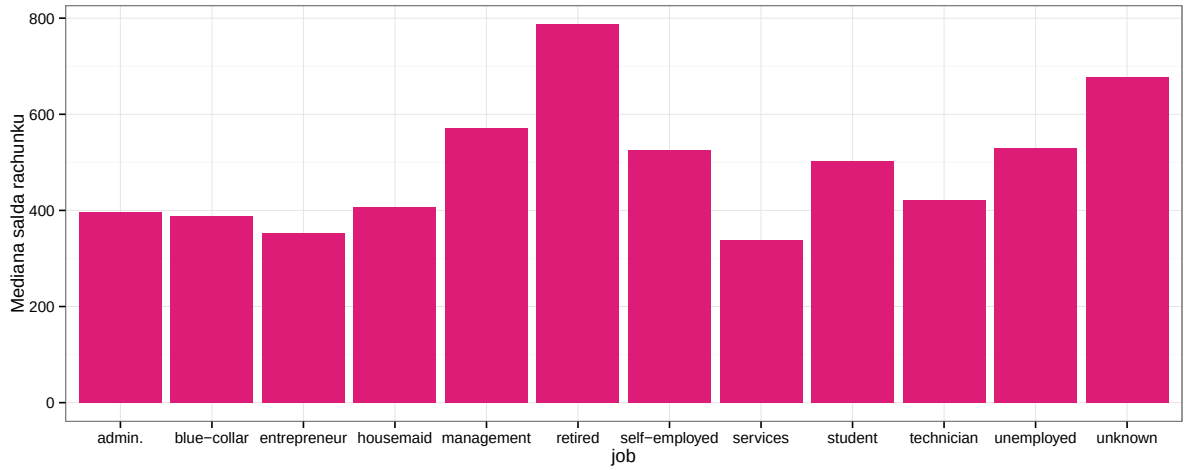
Innym sposobem narysowania powyższych wykresów jest przetworzenie oryginalnych danych do postaci tabeli, w której każdej kategorii będzie odpowiadała liczebność, częstość itd. Pewną przewagą

tego podejścia jest to, że możemy narysować bardziej skomplikowane wykresy słupkowe — pierwsze podejście na to nie pozwala. Poniżej podaje odpowiedni skrypt, ale nie zamieszczam wykresów, gdyż są one niemal identyczne z otrzymanymi powyżej. I jeszcze ostatnia, bardzo ważna uwaga. Ponieważ domyślnie `geom_bar()` zlicza wystąpienia, bo jest tam `stat="bin"`, wiec musimy mu powiedzieć, że ma tego nie robić, bo sami już o to zadbaliśmy. Dlatego wewnątrz tej funkcji należy napisać `stat="identity"`.

```
> ## Dane: bank, przekształcamy ze względu na zmienną education
> dfbar <- bank %>%
+    group_by(education) %>%
+    summarize(Liczebnosc = n()) %>%
+    mutate(Czestosc = Liczebnosc/sum(Liczebnosc))
> dfbar

Source: local data frame [4 x 3]

  education Liczebnosc Czestosc
1   primary       6851   0.1515
2 secondary      23202   0.5132
3  tertiary      13301   0.2942
4   unknown       1857   0.0411

> p1 <- ggplot(dfbar, aes(x=education, y=Liczebnosc)) + theme_bw()
> p1 + geom_bar(fill="#dd1c77", stat="identity") # musi być "identity"
> p2 <- ggplot(dfbar, aes(x=education, y=Czestosc)) + theme_bw()
> p2 + geom_bar(fill="#dd1c77", stat="identity")
```

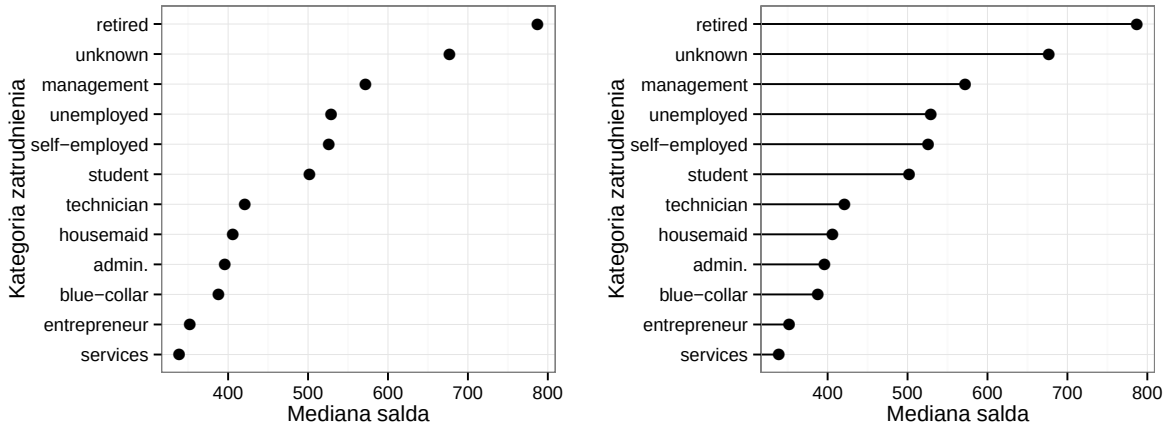
Powyższa strategia pozwala tworzyć wykresy, w których wysokość słupka jest wartością zagregowanej zmiennej ciągłej. Jako miarę agregacji można wziąć średnią, medianę, kwantyl itd. Zobaczmy jak kształtuje się mediana salda rachunku w zależności do kategorii zatrudnienia.

```
> ## Ramka danych: bank, wykres słupkowy dla mediany salda w zależności od zatrudnienia
> balJob <- bank %>%
+    group_by(job) %>%
+    summarize(MedianaSaldo = median(balance))
> head(balJob, 3)

           job MedianaSaldo
1       admin.          396
2  blue-collar          388
3 entrepreneur          352

> ggplot(balJob, aes(x=job, y=MedianaSaldo)) + theme_bw() +
+    geom_bar(fill="#dd1c77", stat="identity") + ylab("Mediana salda rachunku")
```



Jako pewną ciekawostkę (nie będę jej omawiał) i zarazem alternatywę dla wykresu słupkowego — gdy liczba kategorii zmiennej jest duża — przedstawiam wykres punktowy. Dodatkowo kategorie zmiennej zostały posortowane ze względu na saldo rachunku, a na drugim wykresie zostały dodane linie wiodące. W `geom_segment()` musimy podać gdzie się kończą x i y, bo swój początek mają w punkcie.

```
> ## Wykres punktowy z posortowanymi kategoriami (reorder)
> p <- ggplot(balJob, aes(x=MedianaSaldo, y=reorder(job, MedianaSaldo))) + theme_bw()
> p <- p + geom_point(size=3) + ylab("Kategoria zatrudnienia") + xlab("Mediana salda")
> p  # wykres lewa strona
> p + geom_segment(aes(yend=job), xend=0) #dodaj linie wiodące
```

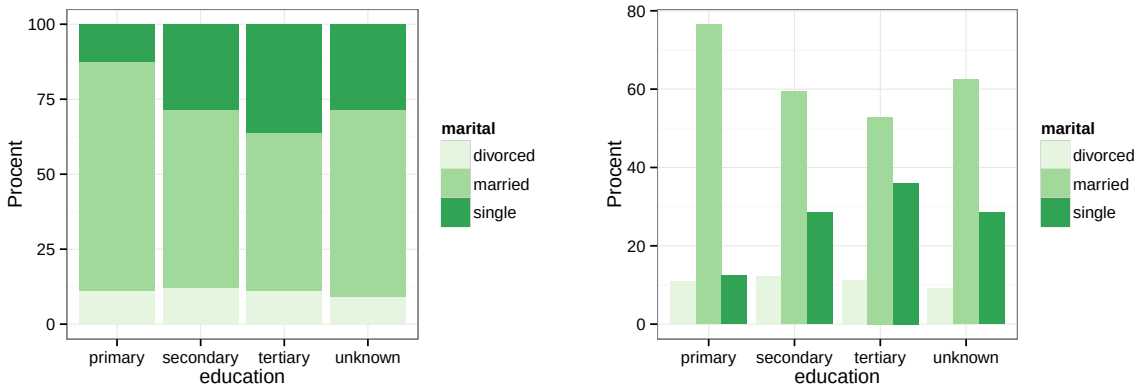


Pozostaje jeszcze omówić możliwość dołączania kolejnej zmiennej dyskretnej, której poziomy znajdą sie w tzw. legendzie. Modyfikacje poprzednich przykładów będą niewielkie. Pierwsza z nich — dość oczywista, bo wymagająca wskazania, która zmienna ma być w legendzie (element estetyczny). Odbywa się to poprzez umieszczenie w `aes` kolejnego elementu `fill=nazwa_zmiennej`. Druga modyfikacja — ma charakter wizualny i odnosi się do zmiany domyślnego ustawiania słupków jeden na drugim (*stack*). Taki wykres nazywamy wykresem słupkowy zestawionym. Jeśli chcemy otrzymać wykres słupkowy zgrupowany (słupki pojawią się obok siebie), wtedy wpiszemy `geom_bar(position="dodge")`.

W przykładzie ilustrującym pokażemy, jak rozkładają się wartości procentowe dla zmiennej stan cywilny (`marital`) w zależności do wykształcenia (`education`). Konstruując tabelę, stanowiącą dane wejściowe dla wykresu, należy pamiętać, które kategorie mają sumować się do 100%. Kolory wybieramy z palety pakietu `RColorBrewer`.

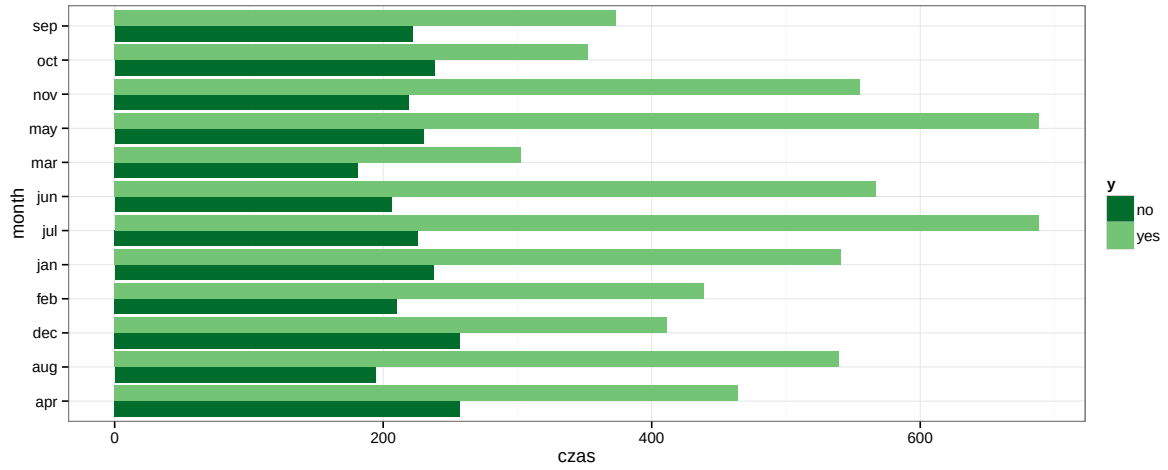
```
> ## Ramka danych: bank; budujemy ramkę: edukacja a stan cywilny
> ## procenty sumują się do 100 w obrębie marital
```

```
> eduMarit <- bank %>%
+    group_by(education, marital) %>%
+    summarize(Liczebnosc = n()) %>%
+    mutate(Procent = 100*Liczebnosc/sum(Liczebnosc))
> eduMarit <- as.data.frame(eduMarit)
> head(eduMarit, 3)

  education  marital Liczebnosc Procent
1   primary divorced        752    11.0
2   primary  married       5246    76.6
3   primary   single        853    12.5

> p <- ggplot(eduMarit, aes(x=education, y=Procent, fill=marital)) + theme_bw()
> p + geom_bar(stat="identity")+ scale_fill_brewer(palette="Greens") # position="stack"
> p + geom_bar(stat="identity", position="dodge")+ scale_fill_brewer(palette="Greens")
```



Na stronie 61 utworzyliśmy ramkę danych (`monYCzas`), w której przedstawiliśmy średni czas rozmowy w zależności od miesiąca i zgody na utworzenie lokaty. Zobaczmy, że szybciej zbudujemy wykres wykorzystując pakiet `ggplot2`. Wprowadzimy też kolejną funkcję: `coord_flip()`, która zamienia oś X i Y miejscami — następuje obrót o 90°. Tym razem ręcznie wprowadzimy nazwy kolorów używając `scale_fill_manual(values=...)`.

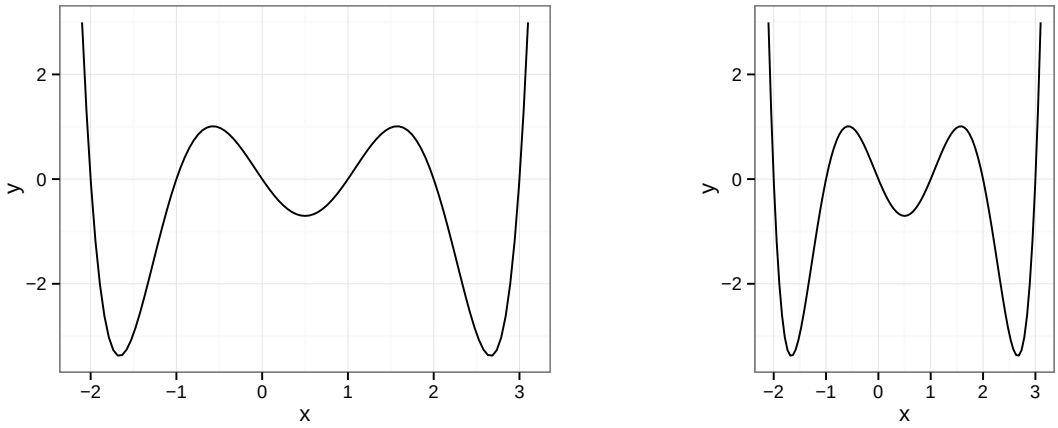
```
> head(monYCzas,3) # przypomnijmy strukturę ramki danych

  month   y czas
1   apr  no  257
2   apr yes  464
3   aug  no  194

> p <- ggplot(monYCzas, aes(x=month, y=czas, fill=y)) + theme_bw()
> p + geom_bar(stat="identity", position="dodge") +
+    scale_fill_manual(values=c("#006d2c", "#74c476")) + coord_flip() #obrót o 90 st.
```



Wykresy rozkładu zmiennej

Rozkład zmiennej można przedstawić w postaci histogramu, funkcji gęstości estymatora jądrowego, dystrybuanty empirycznej i wykresu pudełko-wąsy. Czasami chcemy dodatkowo narysować rozkład teoretyczny i porównać go z empirycznym. Dlatego zaczniemy od funkcji `stat_function(fun = nazwaFunkcji)`. Za jej pomocą narysujemy wykres dowolnej krzywej, przy czym za `nazwaFunkcji` wstawiamy nazwę funkcji już zaimplementowanej w **R** (np.: `sin`, `log`, `dnorm` itp.), lub nazwę funkcji zdefiniowanej przez nas. Zacznijmy od tej drugiej sytuacji. Zadaniem jest narysowanie wykresu wielomianu 6 stopnia, którego postać zapiszemy w obiekcie o nazwie `mojaf`. Pakiet `ggplot2` wymaga, by dane wejściowe były ramką danych. W naszym przykładzie wystarczy podać dwie wartości x, które są tożsame z krańcami przedziału — u nas będzie to $(-2.1, 3.1)$. W poniższym przykładzie zwróćmy uwagę na skalowanie osi. Jeśli chcemy mieć gwarancję, że proporcja odległości na osi X do odległości na osi Y będzie 1:1, wtedy musimy użyć funkcji `coord_fixed(ratio = 1)`.

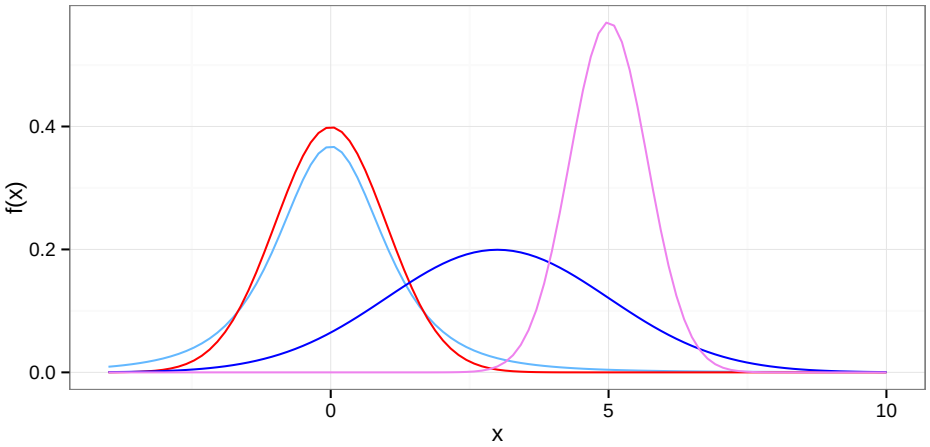
```
> ## Rysowanie wielomianu, o zadanie postaci funkcyjnej
> wyk0 <- ggplot(data.frame(x = c(-2.1, 3.1)), aes(x)) + theme_bw()
> mojaf <- function(x) 0.2*x^6 - 0.6*x^5 - x^4 + 3*x^3 + 0.8*x^2 - 2.4*x
> wyk <- wyk0 + stat_function(fun = mojaf)
> wyk #np. odcinek [0,1] na osi X jest dłuższy od identycznego na osi Y
> wyk + coord_fixed(ratio = 1) #gwarancja, że długości będą identyczne
```



Przejdźmy teraz do narysowania kilku funkcji gęstości. Tym problem zajmowaliśmy się np. na stronie 29. Postępowanie zasadniczo nie różni się od zaprezentowanego powyżej. Jednak zmuszeni jesteśmy do uwzględnienie kolejnego argumentu w funkcji `stat_function()`, gdyż narysowanie gęstości wymaga znajomości parametrów. Dla rozkładu t-Studenta (funkcja gęstości `dt`) musimy podać liczbę stopni

swobody (df), a dla rozkładu normalnego średnią i wariancję, o ile nie chcemy przyjąć wartości domyślnych, tj. odpowiednio 0 i 1. Za to odpowiedzialny jest argument: arg=list(..), gdzie zamiast kropek podajemy nazwy parametrów i ich wartości.

```
> ## Rysowanie różnych funkcji gęstości
> dfX <- data.frame(osX = c(-4, 10)) #przedział zmienności x
> f <- ggplot(dfX, aes(x=osX)) + theme_bw() + xlab("x") + ylab("f(x)")
> f + stat_function(fun = dt, colour = "steelblue1", args=list(df=3)) +
+     stat_function(fun = dnorm, colour = "red") + #normalny N(0,1)
+     stat_function(fun = dnorm, colour = "blue", args=list(mean=3, sd=2)) +
+     stat_function(fun = dnorm, colour = "violet", args=list(mean=5, sd=0.7))
```



Prezentację rozkładów empirycznych rozpoczniemy od histogramu — geom_histogram() i gęstość estymatora jądrowego — geom_density(). W wypadku tej pierwszej funkcji nierzadko domyślne argumenty zmuszeni będziemy zmieniać. Pierwszym z takich argumentów jest binwidth, odpowiadający za szerokość pojedynczego słupka. Jego wartość domyślna jest równa rozstępowi podzielonemu przez 30, czyli range(x)/30. Widzimy więc, że domyślna liczba klas histogramu to 30. Jeśli chcemy to zmienić, wtedy zastąpmy 30 inną liczbą, a wynik dzielenia wpiszemy jako argument funkcji. Oczywiście, im mniejsza binwidth tym więcej klas.

Dla histogramu na osi rzędnej (tzw. Y) przyjmowane są liczebności. Jeśli chcemy na histogram nanieść gęstość estymatora jądrowego, wtedy pojawi sie problem jednostek. W konsekwencji funkcji gęstości nie będzie po prostu widać (podobnie jak: liczebności vs. częstości). Dlatego musimy zmienić skalę dla histogramu z domyślnej (liczebności) na skalę odpowiadającą gęstości. Odbywa się to poprzez dodatkowy argument: aes(y=..density..).

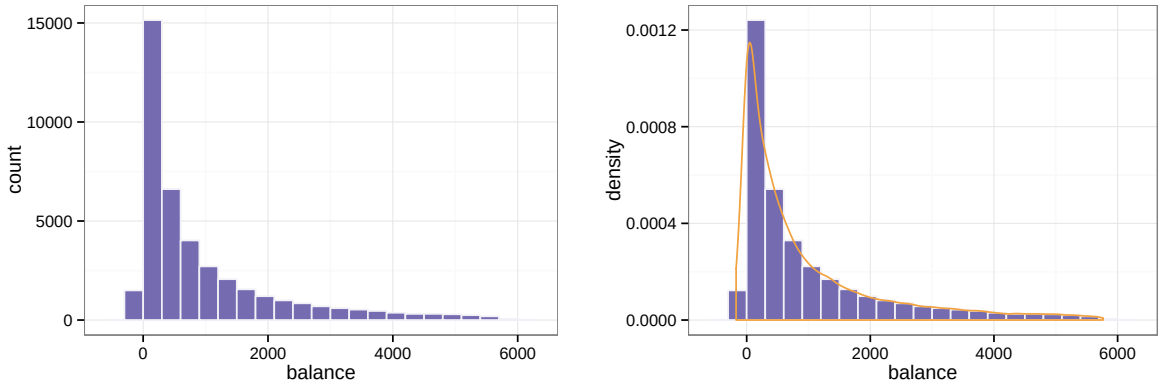
Stwórzmy histogram dla zmiennej saldo rachunku (balance) z ramki danych bank. Ze względów wizualnych ograniczmy się do obserwacji leżących między 5 a 95 percentylem. Nowa ramka będzie miała nazwę: bankCut.

```
> ## Ramka danych: bank; przygotowanie danych: balance między 5 a 95 percentylem
> (quantBalance <- quantile(bank$balance, probs=c(0.05, 0.95)))

  5%  95%
-172 5768

> bankCut <- bank %>%
+    filter(balance > quantBalance[1], balance < quantBalance[2])
```

Przystępując do rysowania histogramu zmienimy domyślną kolorystykę. Argument fill pozwala wypełniać słupki kolorem, natomiast col odpowiada za kolor obramowania słupka. Kod po prawej stronie pozwala nanieść gęstość, gdyż została w nim zmieniona skala na ..density...

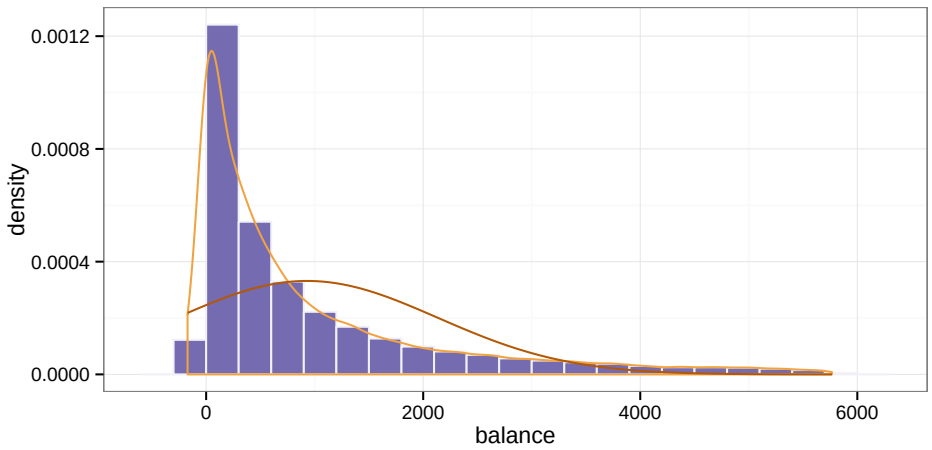
```
> p <- ggplot(bankCut, aes(x=balance)) +
+         theme_bw()
> p  + geom_histogram(
+         fill="#756bb1",
+         col="#f2f0f7",
+         binwidth=300)
```

```
> p <- ggplot(bankCut, aes(x=balance)) +
+         theme_bw()
> p  + geom_histogram(aes(y=..density..),
+         fill="#756bb1",col="#f2f0f7",
+         binwidth=300) +
+       geom_density(col="#f1a340")
```



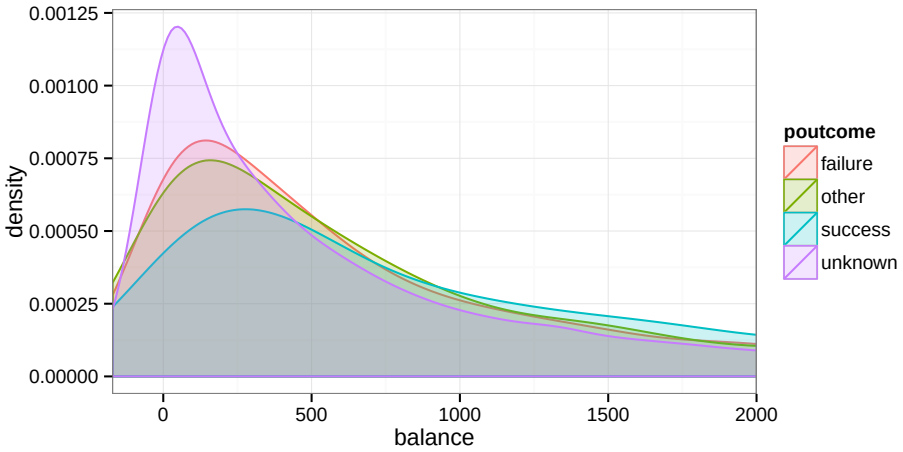
Poniżej ten sam przykład z naniesioną gęstością teoretyczną. Przyjęliśmy, że będzie to rozkład normalny ze średnią 930 i odchyleniem standardowym 1203 obliczonymi z próby. Jak łatwo zauważyć, rozkład empiryczny w ogóle nie przypomina rozkładu normalnego.

```
> p <- ggplot(bankCut, aes(x=balance)) + theme_bw()
> p  + geom_histogram(aes(y=..density..),fill="#756bb1",col="#f2f0f7", binwidth=300) +
+       geom_density(col="#f1a340") +
+       stat_function(fun = dnorm, args = list(mean = 930, sd = 1203), col="#b35806")
```



Funkcja gęstości estymatora jądrowego pozwala w dość prosty sposób na rozszerzenie analizy do porównań gęstości między grupami. Załóżmy, że chcemy zobaczyć, jak wyglądają gęstości ze względu na wynik ostatniej kampanii marketingowej (`poutcome`). Jeśli chcemy wypełnić gęstości kolorem, wtedy za pomocą `fill`, musimy wskazać na zmienną `poutcome`. W tej sytuacji warto zmienić poziom wypełnienia/pokrycia kolorem, sterując parametrem $\alpha \in (0,1)$, np. `geom_density(alpha=0.2)`. Z kolei jeśli chcemy narysować kolorowe funkcje, musimy użyć `color`. A co sie stanie, gdy użyjemy obu naraz? Efekt wizualny będzie milszy dla oka, co pokazuje poniższy przykład.

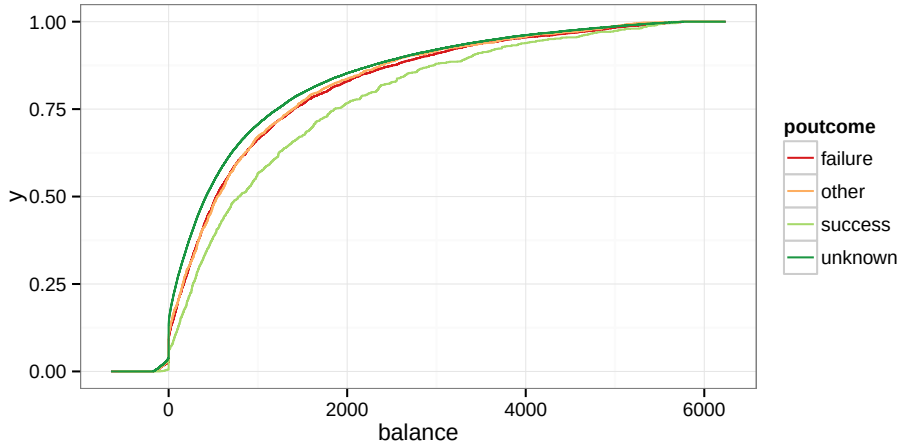
```
> p <- ggplot(bankCut, aes(x=balance, fill=poutcome, color=poutcome)) + theme_bw()
> p + geom_density(alpha=0.2) + #gdy alpha=1, wtedy całkowite wypełnienie (nie widać)
+     coord_cartesian(xlim=c(-172,2000)) #ogranicz x do przedziału
```



Przypomnijmy, że zmiana domyślnego koloru — czy to w sposób manualny, czy poprzez wykorzystanie kolekcji z pakietu `RColorBrewer` — powinna uwzględniać w tym przykładzie element skali (`scale_`) odnoszący się zarówno do `fill` jak i `color`. Dlatego, decydując się na pakiet, należałoby jednocześnie dodać: `scale_fill_brewer(palette=...)` + `scale_color_brewer(palette=...)`.

Dystrybuantę empiryczną narysujemy przy użyciu funkcji: `stat_ecdf()`, co łatwo zapamiętać, gdyż po `stat` mamy skrót od *empirical cumulative density function*. Do jej narysowania używane są linie, więc jeśli chcemy porównać rozkłady w grupach — jak to miało miejsce we wcześniejszym przykładzie — wtedy musimy zadecydować o sposobie ich odróżniania. Naturalnym wyborem będzie użycie `color`, choć teoretycznie `linetype` można też wykorzystać. To ostatnie rozwiązanie polecam jako ćwiczenie, by się samemu przekonać, że zaprezentowany poniżej wariant kolorystyczny jest czytelniejszy.

```
> p <- ggplot(bankCut, aes(x=balance, color=poutcome)) + theme_bw()
> p + stat_ecdf() + scale_color_brewer(palette="RdYlGn")
```

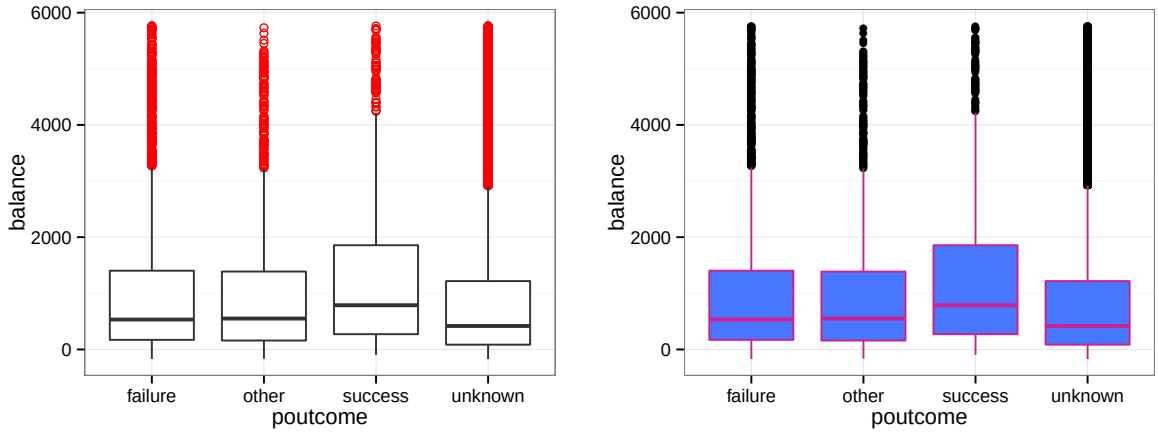


Ostatnią z rozpatrywanych tu funkcji, a pozwalających na ocenę rozkładu zmiennej, jest funkcja `geom_boxplot()`. W definicji `ggplot()` oś X reprezentowana jest przez zmienną dyskretną, natomiast oś Y przez zmienną ciągłą — nie odwrotnie. Jeśli mamy życzenie zamienić osie, wtedy użyjmy wspomnianej wcześniej funkcji `coord_flip()`.

Wracając do funkcji rysującej wykres pudełko-wąsy nadmieńmy, że możemy jej użyć bez żadnych parametrów (zostaną użyte domyślne). Jednak znajomość przynajmniej niektórych bywa przydatna. Poniżej kontynuacja poprzedniego przykładu. Zamieszczam dwa wykresy, w każdym zwracając uwagę na inne elementy formatowania.

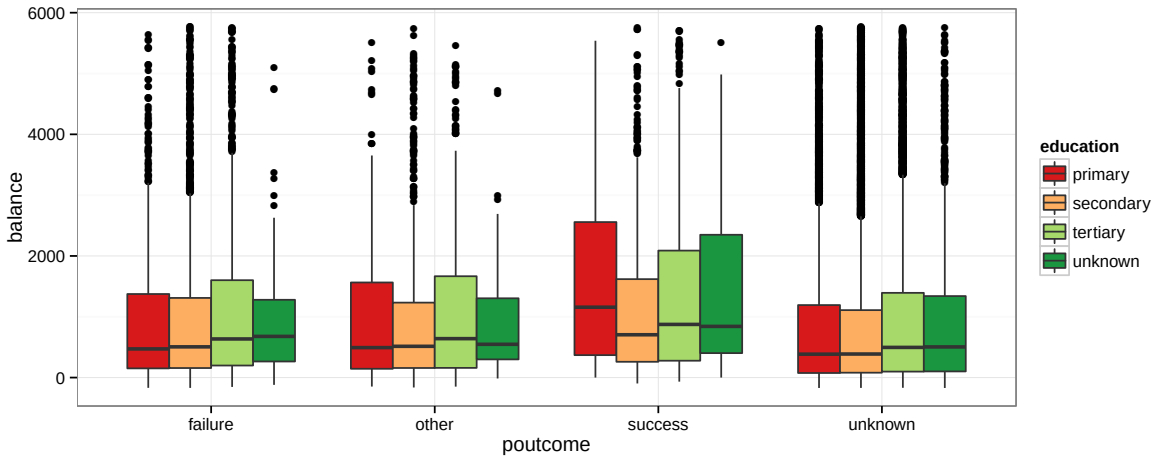
```
> p <- ggplot(bankCut, aes(x=poutcome, y=balance)) + theme_bw()
> p + geom_boxplot(outlier.colour = "red", # dla obserwacji odstających: kolor oraz
```

```
+                   outlier.shape = 1,       #kształt
+                   outlier.size = 2)        #rozmiar
> p + geom_boxplot(fill="royalblue1", col="violetred")  #wypełnienie i obramowanie
```



Na zakończenie zbudujmy ten sam wykres wykorzystując dodatkowy podział ze względu na edukację (`education`) — ta zmienna będzie w legendzie.

```
> wyk <- ggplot(bankCut, aes(x=poutcome, y=balance, fill=education)) + theme_bw()
> wyk <- wyk + geom_boxplot() + scale_fill_brewer(palette="RdYlGn") #kolory z palety
> wyk
```

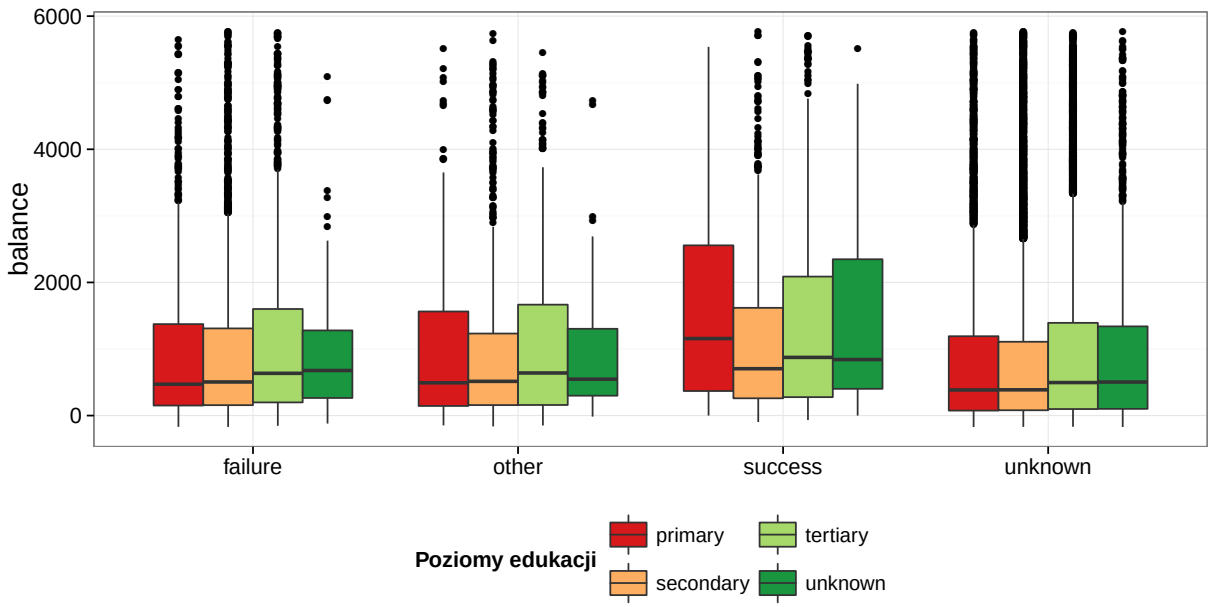


Zaawansowane formatowanie

Zamieszczony materiał w tej części polecam tylko tym, którzy odczuwają niedosyt, w zakresie przedstawionych do tej pory możliwości formatowania. Komentarze ograniczę do minimum — niech to będą „przepisy". Na przykładzie wykresu pudełko-wąsy pokażemy, w jaki sposób można formatować legendę.

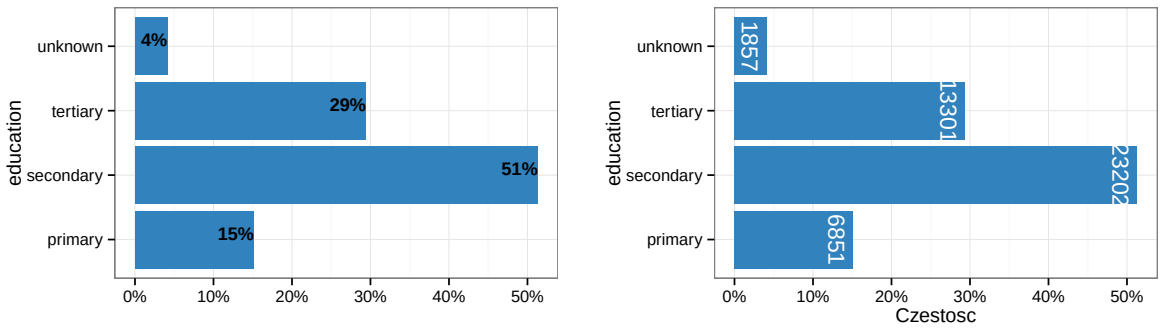
```
> ## Wykorzystuję poprzedni wykres: wyk
> library(grid) #potrzebne, aby użyć: unit
> wyk + theme_bw(base_size=16) + #podstawowy rozmiar czcionki
+       labs(fill="Poziomy edukacji") + #Zmiana nazwy legendy
+       theme(legend.position="bottom", #pozycja legendy (left, right, top, bottom)
+             plot.margin = unit(c(0.1,0.1,-0.4,0), "cm"), #przycina marginesy wykresu
+             legend.key.size=unit(2, "line"), #powiększa elementy legendy
+             legend.key=element_blank(), #usuwa ramki legendy
+             axis.title.x=element_blank(), #usuwa podpis pod osią ox
```

```
+            legend.text=element_text(size=12)) + #rozmiar tekstu legendy
+       guides(fill = guide_legend(nrow = 2)) #legenda w 2 wierszach (ładniej 1 wiersz)
```



Kolejne zadanie zmierza do naniesienia wartości procentowych na słupki. Wszelkie adnotacje tego typu (czy jakikolwiek tekst) nanosimy na wykres przy użyciu funkcji `geom_text()`. Poza tym chcemy, aby na jednej osi były wartości procentowe (ze znakiem %) — musimy zmienić etykiety ciągłej osi *Y* funkcją `scale_y_continuous(labels=...)`. Wykorzystajmy stworzoną na str. 69 ramkę danych `dfbar`.

```
> library(scales) #potrzebny do procentów na osi
> dfbar #przypomnienie struktury

Source: local data frame [4 x 3]

  education Liczebnosc Czestosc
1   primary       6851   0.1515
2 secondary      23202   0.5132
3  tertiary      13301   0.2942
4   unknown       1857   0.0411

> p <- ggplot(dfbar, aes(x=education, y=Czestosc)) + scale_y_continuous(labels=percent)
> p <- p + geom_bar(fill="#3182bd", stat="identity") + theme_bw() + coord_flip()
> p + geom_text(aes(y=Czestosc, label=sprintf("%.0f%%", 100*Czestosc)), size=4,
+     hjust=1, vjust=0, fontface="bold") + ylab("") #vjust i hjust (poziom i pion)
> p + geom_text(aes(y=Czestosc, label=Liczebnosc), vjust=1.5, angle=-90, col="white")
```

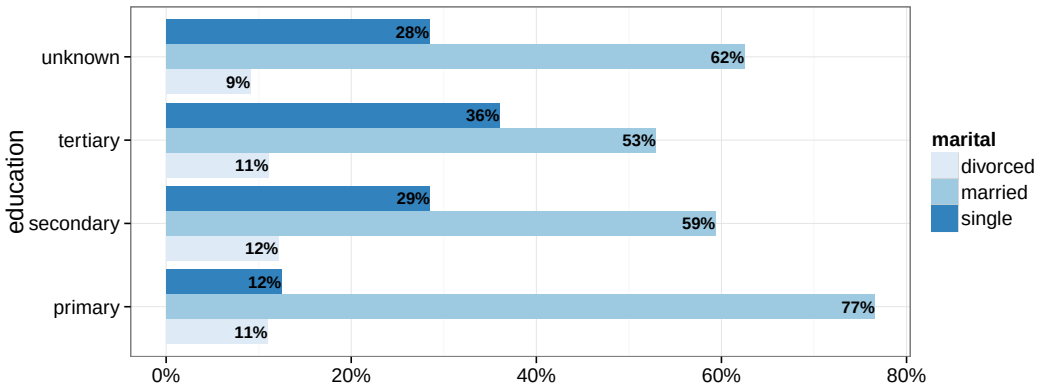


Zadanie jak powyżej, przy czym „dokładamy” zmienną `marital`, która znajdzie się w legendzie. Przygotujmy dane do zbudowania wykresu.

```
> ## Przygotowanie danych
> eduMarit <- bank %>%
+    group_by(education, marital) %>%
+    summarize(Liczebnosc = n()) %>%
+    mutate(Czestosc = Liczebnosc/sum(Liczebnosc))
> head(eduMarit,4)

  education  marital Liczebnosc Czestosc
1   primary divorced        752    0.110
2   primary  married       5246    0.766
3   primary   single        853    0.125
4 secondary divorced       2815    0.121
```

Aby poprawnie wyświetlić etykiety procentowe na wykresie zgrupowanym, trzeba użyć argumentu `position_dodge`. Jest to konsekwencja zapisu: `geom_bar(position="dodge")` pozwalającego na pojawienie się słupków obok siebie.

```
> p <- ggplot(eduMarit, aes(x=education, y=Czestosc, fill=marital)) + ylab("")
> p <- p + geom_bar(stat="identity", position="dodge") + coord_flip() +
+     scale_y_continuous(labels=percent) + scale_fill_brewer() + theme_bw(base_size=16)
> p + geom_text(aes(ymax=Czestosc, label=sprintf("%.0f%%", 100*Czestosc)),
+               position=position_dodge(0.9), size=4, hjust=1, fontface="bold")
```



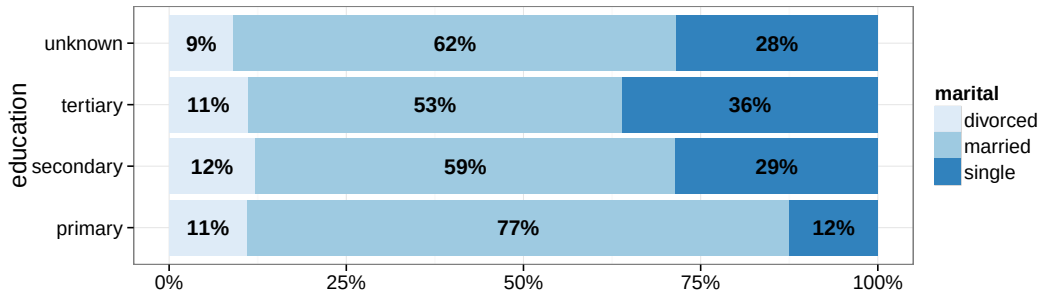
Sytuacja nieznacznie się komplikuje, jeśli chcemy zbudować wykres słupkowy zestawiony. Trzeba zadbać o prawidłowe obliczenie pozycji etykiety. Poniżej przedstawiam takie obliczenia, a pozycja jest zapisana w zmiennej `PozycjaEty`.

```
> eduMarit2 <- eduMarit %>%
+    group_by(education) %>%
+    mutate(PozycjaEty=cumsum(Czestosc) - Czestosc/2)
> head(eduMarit2, 4)

  education  marital Liczebnosc Czestosc PozycjaEty
1   primary divorced        752    0.110     0.0549
2   primary  married       5246    0.766     0.4926
3   primary   single        853    0.125     0.9377
4 secondary divorced       2815    0.121     0.0607
```

Po nieznacznej modyfikacji poprzedniego kodu mamy[5]:

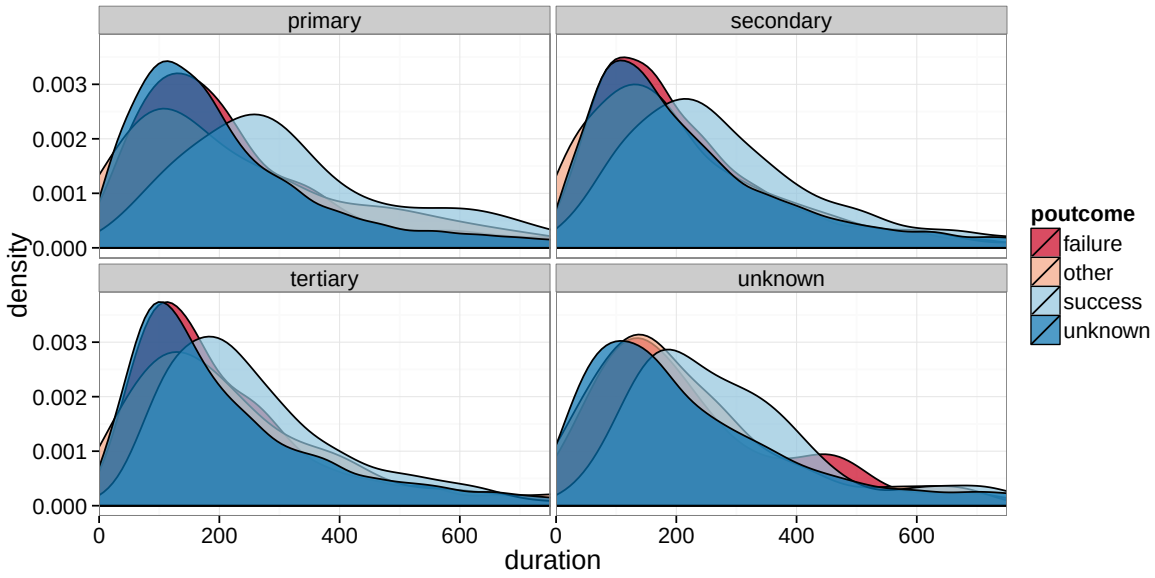
[5]Funkcja `sprintf()` zwraca sformatowany tekst, będący kombinacją tekstu i liczb. Jest to tzw. nakładka (*wrapper*) na funkcję `sprintf` języka C.

```
> p1 <- ggplot(eduMarit2, aes(x=education, y=Czestosc, fill=marital))
> p1 <- p1 + geom_bar(stat="identity") + theme_bw(base_size=16) + coord_flip() +
+     scale_y_continuous(labels=percent) + scale_fill_brewer()
> p1 + geom_text(aes(y=PozycjaEty, label=sprintf("%.0f%%", 100*Czestosc)),
+     size=5, fontface="bold") + theme(axis.title.x=element_blank())
```



Na początku rozdz. 5.3 wspomnieliśmy o elemencie *facet*, który pozwala budować wykresy w tzw. panelach. Zademonstrujemy wykorzystanie tego elementu. W tym celu użyjemy funkcji `facet_wrap()`.

```
> wyk <- ggplot(bankCut, aes(x=duration, fill=poutcome)) + theme_bw(base_size=16) +
+       facet_wrap(~education, ncol=2) #użyj: scales="free_y" lub "free_x" lub "free"
> wyk + geom_density(alpha=0.7) + scale_fill_brewer(palette="RdBu")  +
+     coord_cartesian(xlim=c(0,750))
```



5.4. Zadania

Wykorzystując dane `AutoSprzedam.dat` sformułować kilka pytań badawczych. Odnosząc się do nich, zaproponować plan analizy danych i zrealizować go, wykorzystując zawarte w tym rozdziale procedury. Podstawą oceny będzie wnikliwość poznawcza, stopień opanowania zagadnień z zakresu eksploracyjnej analizy danych.

Rozdział 6

Estymacja i testowanie hipotez

Pojawi się później.

Bibliografia

[1] Bache, K. and Lichman, M. (2013), 'UCI machine learning repository'.
URL: *http://archive.ics.uci.edu/ml*

[2] Chang, W. (2013), *R Graphics Cookbook*, Oreilly and Associate Series, O'Reilly Media, Incorporated.

[3] Crawley, M. (2012), *The R Book*, Wiley.

[4] Gągolewski, M. (2014), *Programowanie w języku R*, Wydawnictwo Naukowe PWN.

[5] Maindonald, J. and Braun, W. (2010), *Data Analysis and Graphics Using R: An Example-Based Approach*, Cambridge Series in Statistical and Probabilistic Mathematics, Cambridge University Press.

[6] Matloff, N. and Matloff, N. (2011), *The Art of R Programming: A Tour of Statistical Software Design*, No Starch Press.

[7] Muenchen, R. A. (2011), *R for SAS and SPSS Users*, Springer Series in Statistics and Computing, Springer.

[8] Murrell, P. (2011), *R Graphics, Second Edition*, Chapman & Hall/CRC the R series, Chapman & Hall/CRC Press, Boca Raton, FL.

[9] Spector, P. (2008), *Data Manipulation with R*, Springer.

[10] Wickham, H. (2009), *ggplot2: Elegant Graphics for Data Analysis*, Springer.