KRÓTKIE WPROWADZENIE DO PAKIETU R

Materiał wspomagający przedmiot: Statystyka dla inżynierów

Robert Kapłon

ver. 2.0
(kompilacja: 1 listopada 2021)

Spis treści

Wstęp

Krótkie wprowadzenie do pakietu R ma ci pomóc w poznaniu tego fantastycznego środowiska. Materiałów do nauki R nie brakuje. Znajdziesz bardzo wiele pozycji książkowych, tutoriali czy nawet filmów instruktażowych. Jednak problem osoby początkującej zawsze sprowadza się do pytania: co wybrać. I właśnie dlatego napisałem to wprowadzenie specjalnie dla ciebie. Jeżeli opanujesz umiejętności na jego poziomie, to z powodzeniem wykonasz nawet złożone analizy. Założyłem tym samym, powiesz później czy słusznie, że ma być ono twoim jedynym źródłem wiedzy na tym przedmiocie. Jeżeli jakiś treści nie rozumiesz i musisz szukać pomocy np. w Internecie, to dla mnie jest to sygnał, że ten fragment lub fragmenty muszę jeszcze dokładniej opisać. Nie zwlekaj i napisz mi o tym. Mój adres to: `robert@pwr.edu.pl`.

Mniej świadomość, że wprowadzenie nie aspiruje do miana kompletnego i wyczerpującego. Wiele istotnych treści pominąłem, ze wszech miar celowo, włączając w to nawet podstawowe zagadnienia. Nie chcę wywołać u ciebie efektu przytłoczenia. Oczywiście, jak skończysz pracę z tym wprowadzeniem, możesz sięgnąć do szerszych opracowań.

Pisząc, przyjąłem pewną konwencję omówienia zagadnień. Zawsze ilustruję je fragmentami kodu oraz wynikami jego uruchomienia. W wielu miejscach, a szczególnie przy wizualizacji danych, zamieszczam przykłady różniące się nieznacznie. Jeżeli je przeanalizujesz, szybko dostrzeżesz różnice między użytymi funkcjami i argumentami. Czasami takie podejście uważam za lepsze niż szczegółowy opis. Poza tym nie traktuj wykresów jako takich, które przygotowałem zgodnie ze sztuką. Czasami zdarza się, że wyglądają dziwacznie.

W tekście znajdziesz też 3 rodzaje ramek, a na końcu indeks użytych funkcji. Poniżej przeczysz, co zawiera każda z ramek.

Zapamiętaj 0.1

W takiej ramce zamieszczam informacje o szczególnej ważności. Myślałem nawet, aby nazwać ją tak z przymrużeniem oka: bez tego nie zaliczysz. Ale to sugeruje przymus i uczenie się dla oceny. Mam nadzieję, że jednak chcesz się nauczyć czegoś nowego, czegoś co może ci się przydać w późniejszej pracy zawodowej.

Strefa Eksperta 0.1

To ramka jest przeznaczona dla osób, które chcą się dowiedzieć czegoś nadprogramowo. Dlatego jest nieobowiązkowa.

> **WARTO WIEDZIEĆ**
>
> Wiedza jaką się dzielę za pośrednictwem tej ramki jest ważna, potrzebna i na pewno nie można zaliczyć jej do wiedzy ezoterycznej. Jeśli chcesz opanować R przynajmniej na dobrym poziomie zapoznaj się z nią.

Jaką strategię pracy z wprowadzeniem przyjąć. Musze to jasno napisać: samo czytanie — bez aktywnego, równoczesnego używania programu — jest niewystarczające do zdobycia biegłości w posługiwaniu się R. Dlatego gorąco cię zachęcam do przepisywania zamieszczonych tutaj fragmentów programów, a nie ich kopiowania i wklejania. Wpisując kilka razy tą samą funkcję, po prostu ją zapamiętasz. Polecam ci również przeprowadzanie eksperymentów, gdy podczas lektury nasuną się wątpliwości. Zadawaj pytania (a co by było gdyby), zmieniaj kod i testuj.

W opracowaniu wykorzystuję różne zbiory danych, które znajdują się w katalogu dane.

ROZDZIAŁ 1

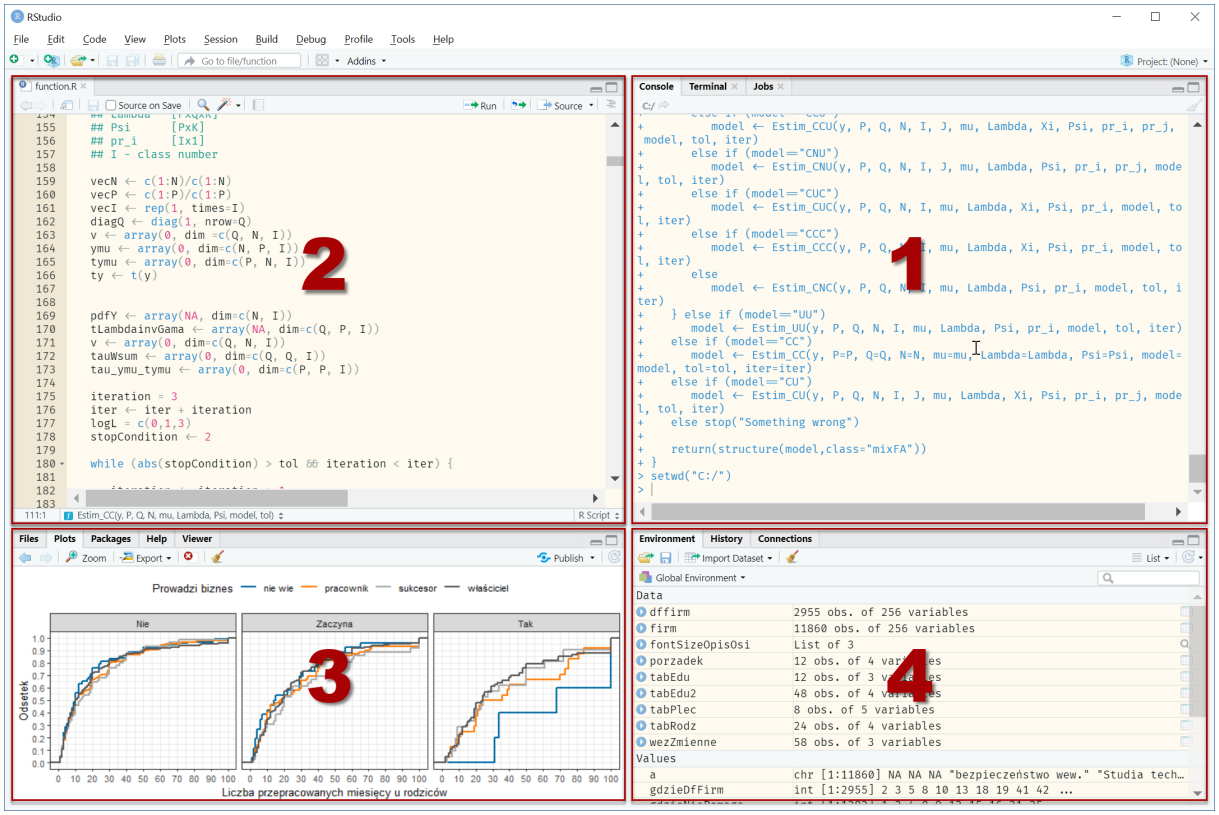
Przygotowanie do pracy w środowisku R

1.1. Instalacja i konfiguracja

Przygotowania rozpoczynamy od ściągnięcia i zainstalowania środowiska **R**. Odwiedź stronę projektu **R**: `http://www.r-project.org`. Z menu po lewej stronie wybierz CRAN oraz lokalizację serwera. Teoretycznie każdy URL powinien zawierać tą samą wersję programu. Pamiętaj o platformie, na której **R** będzie instalowany. Jeżeli wybierzesz *Windows*, to później kliknij na *base* i pobierz plik. Rozważ zmianę lokalizacji, w jakiej program zostanie zainstalowany. Proponuję wybrać `C:\R`. Podczas instalacji wybierz: *Yes (customized startup)*, a później zaznacz pojedyncze okno (*SDI*, a nie domyślne *MDI*).

Od pewnego czasu komunikaty w konsoli **R** są w j. polskim, z kolei pomoc, dokumentacja, tutoriale itp. są w j. angielskim. Taka dwujęzyczność przeszkadza, dlatego będziemy pracować z wersją angielską. Zmian dokonasz, edytując w notatniku plik `Rconsole` — najedź na plik i prawym przyciskiem myszki wybierz: otwórz za pomocą, wskazując np. notatnik (polecam Notepad++). Plik znajduje się w katalogu `etc`. U mnie pełna ścieżka to: `C:\R\R-4.1.1\etc\`. Gdy już otworzysz plik, odszukaj wiersz odnoszący się do definicji języka i dopisz `en`; tak powinien wyglądać uzupełniony wiersz: `language = en`.

Do tworzenia i edytowania skryptów **R** wykorzystamy darmowe, zintegrowane środowisko **RStudio**: `https://www.rstudio.com/products/rstudio/download/preview/`. Zainstaluj go w domyślnej lokalizacji — od tej pory będziemy uruchamiać tylko **RStudio**, którego wygląd widzisz na rysunku.



Całe okno **RStudio** podzielone jest na 4 panele główne, z których każdy ma jeszcze kilka zakładek. Położenie paneli względem siebie możesz skonfigurować, wybierając z menu: *Tools* | *Global Options*. Tam też znajdziesz wszystkie ustawienia. Polecam zmienić następujące:

- **General | Basic** — w *Workspace* odznacz opcję: *Restore .RData into workspace at startup*. Dla *Save workspace to .RData on exit* wybierz *Never*. Jeśli opcja jest zaznaczona (*Always*), wtedy

przy każdym uruchomieniu RStudio ładuje do pamięci wszystkie obiekty (zawartość całego środowiska) poprzedniej sesji. Jest to źródłem wielu problemów początkujących użytkowników.
- **Code | Display** — zaznacz *Show margin* i ustaw na 100. Zaznacz też obie opcje *Highlight R function calls*.

W **Oknie 1** widzisz konsolę R. Pozwala ona na interaktywną pracę. Wpisz w niej `2+2` i naciśnij Enter. Nastąpi natychmiastowa interpretacja i zobaczysz wynik. Jeśli dowolny zapis poprzedzisz znakiem #, to poinformujesz program, że ma go traktować jak komentarz — spróbuj wpisać: `#2+2`. Tryb interaktywny jest użyteczny wtedy, gdy chcesz coś szybko obliczyć lub sprawdzić, czy składnia jest poprawna. Musisz jeszcze wiedzieć, że oprócz widocznego znaku > czasami będzie pojawiał się znak plusa. Program wtedy czeka na dokończenie składni. Wykonaj działanie odejmowania dwóch liczb (5−4), ale w następujących krokach: wpisz `5 -`, naciśnij enter, a następnie wpisz 4. Gdy chcesz wyjść z trybu oczekiwania, naciśnij klawisz `Esc`.

Tryb interaktywny nie sprawdza się nawet w wypadku bardzo krótkich analizy. Dlatego cały skrypt z naszą analizą będziemy tworzyć w **Oknie 2**. Więcej o tworzeniu, edycji i uruchamianiu skryptów znajdziesz w rozdziale 1.2, w którym nauczysz się tworzyć projekty.

W **Oknie 3** mamy kilka zakładek. Omówię dwie: pakiety (*packages*) i pomoc (*help*). Pakiety są bardzo mocną stroną R. Tworzy je grono entuzjastów i miłośników zarówno samego języka jak i również szeroko pojętej analizy danych. Oferują one funkcjonalność niespotykaną w innych pakietach statystycznych. Wyobraź sobie, że wiele pomysłów na analizę danych, które być może przyjdą ci do głowy, zostało już zaimplementowanych właśnie w postaci pakietów. Mamy do dyspozycji 18314 pakietów, które są dostępne w repozytorium CRAN na stronie R. Jeśli chcesz z któregoś skorzystać, to musisz go najpierw zainstalować. Możesz to zrobić na dwa sposoby. Pierwszy — z zakładki *Packages* wybierasz *Install*, a następnie wpisujesz nazwy interesujących cię pakietów. Drugi — w konsoli lub skrypcie uruchamiasz funkcję `install.packages("tutaj_nazwa_pakietu")`. Niezależnie od wyboru tę czynność wykonujesz tylko raz. Pamiętaj jednak, że jeśli takiego pakietu chcesz użyć, wtedy przy każdym uruchomieniu programu RStudio musisz go wczytać za pomocą funkcji: `library("tutaj_nazwa_pakietu")`. Zazwyczaj pakiety ładujemy na samym początku pisanego skryptu. W poniższym przykładzie instaluję pakiet `dplyr`, a następnie go wczytuję.

```
> install.packages("dplyr") # Instalujesz tylko raz
> library(dplyr) # Wczytujesz za każdym razem, gdy chcesz użyć (w nowej sesji RStudio)
```

Może cię zainteresować jeszcze jedna funkcja, która podaje pewne informacje o pakietach. Poniżej, w pierwszym wariancie pokazuje ona listę wszystkich zainstalowanych pakietów na komputerze. Drugi z kolei wyświetla nazwy tych pakietów, które zostały już wczytane do R. Zanim zadasz sobie pytanie: dlaczego R nie widzi jakiejś funkcji, wchodzącej w skład pakietu, sprawdź, czy pakiet został wczytany.

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

> (.packages()) # pokaż, które z zainstalowanych pakietów zastały już wczytane
 [1] "nortest"   "Hmisc"     "Formula"   "survival"  "lattice"   "scales"    "ggthemes"
 [8] "e1071"     "bindrcpp"  "forcats"   "dplyr"     "purrr"     "readr"     "tidyr"
[15] "tibble"    "tidyverse" "ggplot2"   "stringr"   "tools"     "knitr"     "stats"
[22] "graphics"  "grDevices" "utils"     "datasets"  "methods"   "base"
```

Przejdę teraz do omówienia pomocy, którą widzisz w jednej z zakładek **Okna 3**. Możesz tę pomoc przeglądać, albo w okienku wyszukiwarki wpisać nazwę funkcji czy też frazę. Przyznam, że szukając informacji postępuję trochę inaczej i wydaje mi się, że szybciej. Wpisuję w konsoli nazwę interesującej mnie funkcji, poprzedzając ją znakiem zapytania. Wtedy automatycznie, w oknie pomocy, pojawia się pełna informacja. Spróbuj wpisać: `?log`. Jeśli nie znasz dokładnej nazwy funkcji, a chcesz wyszukać wszystko, co dotyczy pewnej frazy, użyj podwójnego pytajnika `??`. Może się zdarzyć, że we frazie pojawia się spacja, wtedy użyj cudzysłowów, np. `??"log normal"`.

Pomocy możesz również poszukiwać w dokumentacji i opracowaniach dostępnych 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`

W **Oknie 4** widzisz zawartość tzw. środowiska (*Environment*). Składają się na niego obiekty, które tworzysz w ramach sesji R. Oprócz listy z nazwami tych obiektów, masz informację o ich rodzaju i rozmiarze. Jeżeli klikniesz na obiekt o strukturze tablicy, to pojawi się podgląd zawartości w postaci tabeli, jaką znasz z arkuszy kalkulacyjnych.

W tej części użyłem dwóch bardzo ważnych słów, które nie zdefiniowałem — obiekt i funkcja. W dalszej części dowiesz się o nich więcej. Teraz zapamiętaj dwie fundamentalne reguły R.

Zapamiętaj 1.1

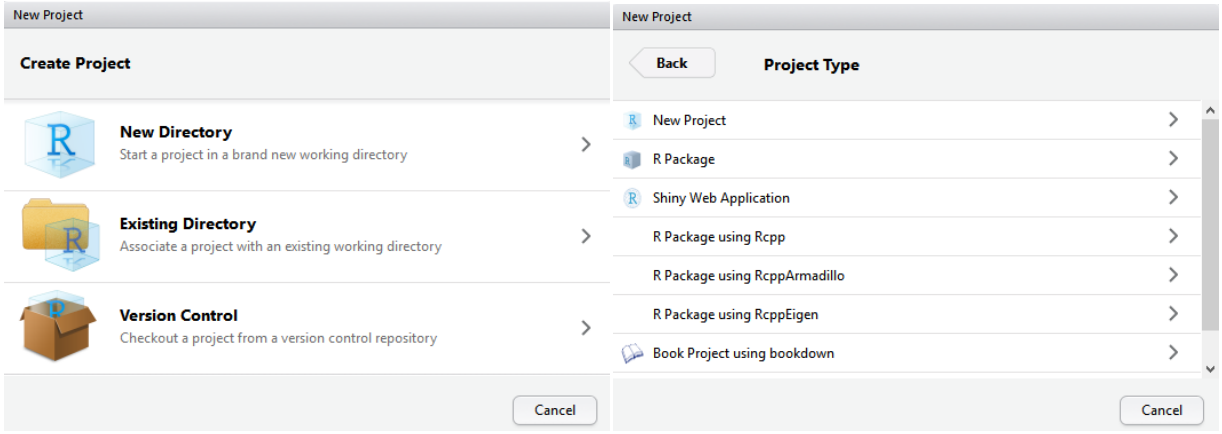
Dwa fundamenty R

- Wszystko w R jest obiektem.
- Wszystko co dzieje się w R, jest następstwem wywołania funkcji.

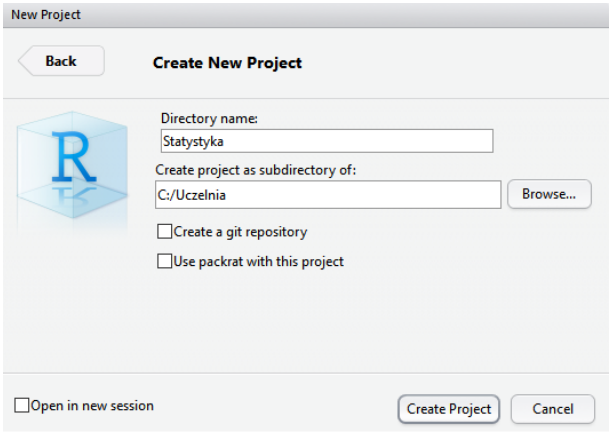
1.2. Tworzenie projektu

Najlepszym sposobem organizacji pracy w RStudio są projekty. Projekt jest tak naprawdę katalogiem, w którym przechowujemy pliki i inne katalogi. Pozwala nam to uporządkować pracę. Możemy utworzyć tam katalog o nazwie `dane` i przechowywać w nim wszystkie zbiory danych, które wykorzystujemy w analizie. W innym katalogu możemy przechowywać pliki graficzne wykresów, a w kolejnym raporty i prezentacje. W konsekwencji tzw. przestrzenią roboczą jest katalog projektu, a nie domyślnie ustalona przez R lokalizacja na dysku. Zapytaj program o tę lokalizację wpisując w konsoli `getwd()`.

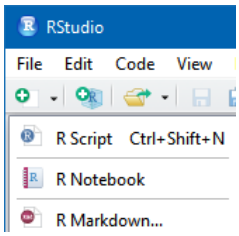
Stwórzmy przykładowy projekt i nazwijmy go: *Statystyka*. Z menu **File | New Project** wybierz: *New Directory*, a następnie *New Project*. Ten etap prezentuję na poniższych oknach.



W kolejnym kroku podaj nazwę projektu (*Statystyka*) i wybierz jego lokalizację na dysku, np. `"C:/Uczelnia"`.



Od teraz całą zawartość katalogu *Statystyka* możesz przenosić między komputerami (sprawdź ponownie lokalizację przestrzeni roboczej). Aby uruchomić **RStudio** i otworzyć projekt, wystarczy że klikniesz na plik `Statystyka.Rproj`, który znajduje się w tym katalogu. Nie ma w nim chyba najważniejszych plików — skryptów z twoimi programami. Aby taki utworzyć, kliknij na ikonkę zielonego plusa i wybierz `R Script`, jak podpowiada ci poniższy rysunek.



Plik zapisz pamiętając o rozszerzeniu: po kropce dodaj dużą literę R. Zauważ, że domyślna lokalizacja zapisu to właśnie twój projekt. Ja utworzyłem plik o nazwie: `zad_rozdz1.R`. W zależności od potrzeby możesz utworzyć wiele takich plików. Od tej pory każdą, nawet najmniejszą analizę, będziemy zapisywać w edytorze plików (**Okno 2**).

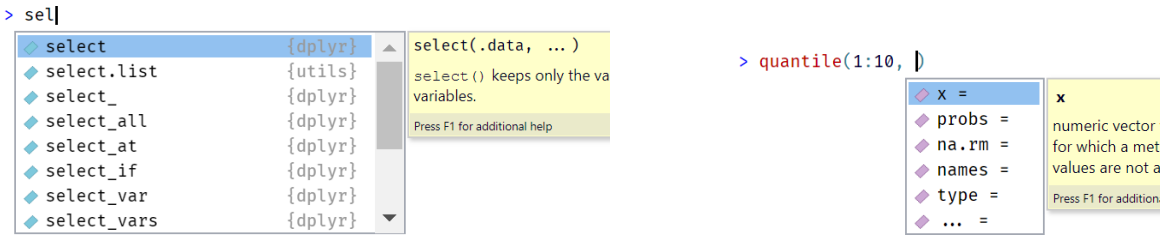
W edytorze plików możesz wpisać cokolwiek i dopóki nie prześlesz tego do konsoli, edytor jest zwykłym notatnikiem. Zatem napisz `2+2` i użyj kombinacji klawiszy: `CTRL+Enter`. To ona przekazuje całą aktywną linię do konsoli — nie ma znaczenia, w której kolumnie jest kursor. Jeśli natomiast chcesz przekazać więcej linii, to zaznacz je i wtedy użyj tego skrótu. Z innymi, dostępnymi skrótami klawiaturowymi możesz zapoznać się wybierając z menu `Help` i `Keyboard Shortcuts`.

Zapamiętaj 1.2

Skróty klawiaturowe

- `Ctrl+ENTER` — wiersz edytora skryptów jest przekazywany do konsoli, a następnie wykonywane są zawarte w nim polecenia. Działa również, jeśli zaznaczysz kilka wierszy.
- `Tab` — pokazuje argumenty funkcji; wybierasz je za pomocą strzałek i Entera.
- ↑/↓ — użycie strzałek w konsoli przywołuje wcześniej napisane komendy.

Program **RStudio** wspomaga cię w trakcie pisania kodu. Zacznij pisać nazwę funkcji i już po trzech znakach — jak na załączonym poniżej lewym rysunku — pojawia się lista. Strzałkami wybierasz właściwą i wciskasz `Enter`. Sugeruję takie podejście, gdyż ono wyklucza błędy pisowni. Chyba najlepszym przykładem jest funkcja, która podaje długość wektora `length()`. Bardzo często ten angielski wyraz jest pisany błędnie.



Kolejnym udogodnieniem są podpowiedzi argumentów funkcji. Zobacz na rysunek po prawej stronie. Zapisałem funkcję `quantile()` i będąc wewnątrz nawiasów wcisnąłem klawisz `Tab`. Pojawiła się lista rozwijana z dodatkowymi argumentami funkcji. Każdy argument jest też opisany w polu na żółtym tle.

Strefa Eksperta 1.1

Skróty klawiaturowe

Listę skrótów klawiaturowych dostępnych w **RStudio** wyświetlisz wciskając kombinację `Shift+ALT+K`. Oto kilka propozycji, które usprawnią pracę z kodem:

- `ALT+↑/↓` — przenosi całą linię tekstu, w której znajduje się kursor, do góry/dołu.
- `Shift+ALT+↓` — kopiuje całą linię, w której znajduje się kursor
- `ALT+-` — wstawia operator przypisania .
- `Shift+Ctrl+M` — wstawia tzw. operator potoku `%>%` (później często będzie używany)

ROZDZIAŁ 2

Struktury danych

2.1. Typy fundamentalne

Liczby są fundamentalnymi obiektami w R. Zapewne znasz podział na liczby: naturalne, całkowite, wymierne, niewymierne, rzeczywiste i zespolone. W pracy ograniczymy się do **typu całkowitego** oraz **rzeczywistego** (typ zmiennoprzecinkowy o podwójnej dokładności). Zapamiętaj, że w liczbach rzeczywistych będziemy oddzielać kropką części całkowite od dziesiętnych, np. 2.77.

Kolejnym obiektem podstawowym są łańcuchy znaków, zwane również napisami, o których mówimy, że są **typu znakowego**. Wykorzystujemy je do przechowywania informacji tekstowych, które ujmujemy w cudzysłowy, np. `"a"` czy `"zgadzam się"`. Wyprzedzając twoje pytanie o możliwość użycia pojedynczych cudzysłowów napiszę: nie ma większego znaczenia czego użyjesz. Ja przyzwyczaiłem sie do podwójnych, gdyż praca z tekstem pisanym w j.angielskim zmusza do uwagi tam, gdzie pojawia się apostrof. Spróbuj w konsoli R wpisać zdanie: `"robert's intro."`, a następnie zamień cudzysłowy podwójne na pojedyncze. Dostrzegasz problem?

Czasami chcemy dowiedzieć się, jakiego typu jest obiekt. Pomoże nam w tym funkcja `typeof()`, która zwraca: *integer* dla liczb całkowitych, *double* dla liczb rzeczywistych oraz *character* dla łańcucha znaków. Muszę wspomnieć również o **typie liczbowym** (*numeric*), który odnosi się do dwóch pierwszych typów. Przykładowo funkcja `mode()` zwraca ten typ niezależnie od tego, czy liczba jest całkowita czy rzeczywista. Spójrz na poniższy przykład (na razie nie przejmuj się funkcją `c()` łączącą elementy, bo o niej już niedługo).

```
> c(typeof(7L), typeof(7), mode(7), typeof(1.5), typeof("b. dobrze"))
[1] "integer"   "double"    "numeric"   "double"    "character"
```

Zapewne winny jestem ci wyjaśnienia, bo przecież liczba 7 jest całkowita, a R napisał, że jest rzeczywista (*double*). Okazuje się, że bardzo często liczby które widzimy jako całkowite, R zapisuje i traktuje jak liczby rzeczywiste — robi to celowo. Jeżeli chcesz mieć liczbę całkowitą, to dodaj dużą literę L na końcu jak w powyższym przykładzie.

Obiekty typu **logicznego** mogą przechowywać jedną z dwóch wartości: prawda albo fałsz. W programie za te dwa stany odpowiadają słowa: `TRUE` i `FALSE`. Ponieważ w R wielkość liter ma znaczenie, więc nie zostaną rozpoznane takie słowa jak `true` czy `True` — znane z innych języków programowania. Zapewne wiesz, że określenia słowne (polskie, angielskie) przyjmowane są dla naszej wygody, gdyż komputer przechowuje liczby: 0 i 1 odpowiednio dla fałszu i prawdy. Zapytam z przymrużeniem oka: jeżeli dodasz 3 prawdy, to co otrzymasz? Nasz program jest nad wyraz konsekwentny:

```
> TRUE + TRUE + TRUE
[1] 3
```

Wartości i symbole specjalne uzupełniają naszą listę. Na pewno spotkasz:

- `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.

Przeanalizuj poniższe przykłady, aby zobaczyć, w jakich sytuacjach wartości i symbole mogą się pojawić. Zapewne zauważasz, że kolejny raz użyłem funkcji `c()` złączającej elementy. Co więcej powstały obiekt nazwałem wektorem. Chyba najwyższa pora, abyśmy przeszli do jego opisu. Obiecuję, że już w podrozdziale 2.3 skupimy się na wektorach. Musimy jeszcze zapoznać się z bardzo ważnym zagadnieniem tworzenia obiektów nazwanych, czemu poświęcimy kolejny podrozdział.

```
> log(0) # oblicz logarytm naturalny z 0
[1] -Inf

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

> c(3, 5, 9, NA) # ostatni element wektora traktowany jako brak danych
[1]  3  5  9 NA

> c(5, 3, NULL, 8) # wektor ma 3 elementy (pusty z definicji zostaje pominięty)
[1] 5 3 8
```

WARTO WIEDZIEĆ

Zamiast pytać o typ obiektu, możesz sformułować pytanie bardziej precyzyjne — czy wskazany obiekt jest określonego typu, np. znakowego, rzeczywistego itd. Odpowiedzi udzielą ci funkcje, które zaczynają się przedrostkiem `is`. R daje ci również możliwość rzutowania (konwersji) na inny typy. Załóżmy, że masz wektor postaci: `c("3", "7")`. Zapis mówi, że wektor jest typu znakowego. Ty jednak chcesz mieć liczby. Musisz więc rzutować np. na typ numeryczny za pomocą funkcji `as.numeric(x)`. Jak widzisz, ta grupa funkcji zaczyna sie przedrostkiem `as`. W poniższej tabeli znajdziesz szczegóły.

Tabela 2.1. Sprawdzanie i rzutowanie typów

Typ	Sprawdzenie	Rzutowanie
logiczny	`is.logical(x)`	`as.logical(x)`
całkowity	`is.integer(x)`	`as.integer(x)`
rzeczywisty	`is.double(x)`	`as.double(x)`
numeryczny	`is.numeric(x)`	`as.numeric(x)`
znakowy	`is.character(x)`	`as.character(x)`

2.2. Tworzenie obiektów nazwanych

Do tej pory wpisywaliśmy wyrażenia (w konsoli lub edytorze), które następnie były przetwarzane przez R, a wynik wyświetlany w konsoli, np.

```
> 2 + log(5)
[1] 3.60944
```

Jeżeli wartość tego wyrażenia chcesz wykorzystać później, w jakimś innym miejscu swojej analizy, to musisz go zapisać w pamięci komputera. Aby to zrobić, utwórz obiekt nazwany w dwóch krokach: wybierz nazwę, a następnie wykorzystując operator przypisania `<-`, przypisz tej nazwie obiekt. Zamiast operatora przypisania możesz użyć znaku `=`. Nie polecam takiej zamiany, gdyż prawie wszyscy — z historycznych względów — używają operatora przypisania. Im szybciej się do niego przyzwyczaisz, tym łatwiejszy w interpretacji będzie kod. Wybrałem nazwę `suma` dla obiektu i wpisałem w konsoli:

```
> suma <- 2 + log(5)
```

Być może zastanawiasz się, dlaczego nie widzisz wyniku (jak wcześniej). Otóż R domyślnie nie wyświetla wartości przypisanej do `suma`. Trzeba go do tego zmusić, wpisując nazwę w konsoli. Możemy tak zrobić, gdyż wynik został zapamiętany w pamięci:

```
> suma
[1] 3.60944
```

Powiem ci jeszcze o jednym sposobie wyświetlania obiektów. Ujmij całe wyrażenie w nawisy okrągłe `(suma <- 2 + log(5))`. Przyznam, że podczas pisania programów raczej z tego nie korzystam, chociaż we wprowadzeniu do R od tego nie stronię. Przyświecał mi jeden cel — oszczędność miejsca.

Strefa Eksperta 2.1

Nazwy, obiekty i kopie

Fraza: tworzenie obiektów nazwanych jest pewnym skrótem myślowym i uproszczeniem. Ten proces wygląda następująco: tworzysz obiekt (po prawej stronie), a później wiążesz go z nazwą (po lewej stronie). W ten sposób zapewniasz sobie dostęp do obiektu poprzez nazwę. Obiekt nie ma nazwy, to nazwa wskazuje na ten obiekt. W tym przykładzie

```
> x <- 7:9
> y <- x
```

obie nazwy `x` i `y` wskazują na ten sam obiekt — nie jest on kopiowany. Co się stanie, jeżeli zmienimy `y`, np. `y[1] <- 0`? Jeśli programujesz np. w Pythonie to od razu odpowiesz: `x` również się zmieni. W R działa to inaczej. Przy takich zmianach tworzona jest automatyczna kopia obiektu. W konsekwencji masz już dwa różne obiekty — `x` i `y` rozdzieliły się. Po co o tym piszę? Jeżeli masz duże zbiory danych, a wszystkie operacje wykonujesz w pamięci operacyjnej komputera, to tworzenie kopii jeszcze bardziej te zasoby uszczupla.

Powinienem poruszyć kwestię wyboru nazwy dla obiektu, gdyż nie ma tutaj pełnej dowolności. W R nazwy mogą składać się z ciągu liter, cyfr, kropki, podkreślenia. Nie możesz nazw zaczynać od cyfr, używać znaków specjalnych (np. %, #) oraz słów kluczowych języka R (np. `if`, `else`, `TRUE`, `FALSE`, `NA`). Przykładem poprawnych nazw są: `grupaWiek`, `grupa.wiek`, `grupa_wiek`, `GrupaWiek`. Jak już wiesz, w R jest istotna wielkość liter, dlatego nazwa pierwsza nie jest równoważna nazwie ostatniej.

2.3. Wektory

Wektor jest ciągiem elementów tego samego typu. Dlatego tworząc wektor, nie możesz typów mieszać. Zapewne pamiętasz z podrozdziału 2.1, że do wyboru masz typ: liczbowy, znakowy, logiczny. Jeśli nie zastosujesz się do tego wymogu i zechcesz utworzyć wektor składający się z liczby 5.7 i słowa `"czytaj"`, wtedy R dokona tzw. koercji, czyli ujednolicenia typu. W konsekwencji liczba 5.7 zostanie potraktowana jako łańcuch znaków, a wektor będzie typu znakowego.

2.3.1. Tworzenie wektorów.

Wektory w R utworzysz na kilka sposobów:

1. `c()` — funkcja łącząca (*combine*) elementy, np. `c(4, 5, 3.7)`;
2. `:` — operator tworzący ciąg arytmetyczny o różnicy 1, np. `2:5`;
3. `seq(from, to, by, length.out)` — funkcja tworząca ciąg (*sequence*) arytmetyczny o różnicy `by`, np. `seq(2, 8, 3)` utworzy wektor o wartościach: 2, 5, 8; zamiast `by` możesz podać liczbę elementów, dla naszego przykładu: `length.out = 3`;
4. `rep(x, times, each)` — funkcja powielająca (*replicate*) zadaną liczbę razy: cały obiekt `x` (`times`) albo każdy jego element (`each`).

Jeśli wpiszesz w konsoli R znak zapytania i nazwę funkcji, np: `?seq`, wtedy przejdziesz do pomocy. Zanim jednak zdecydujesz się na ten krok, przeanalizuj poniższe przykłady.

```
> ## 1. 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
[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"
```

```
> ## 2. 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 - użycie c() nie ma uzasadnienia
[1]  3  4  5  6  7  8  9 10

> 7:2
[1] 7 6 5 4 3 2

> c(3:10, 4:2, -3) # tutaj jest sens, bo łączymy 3 wektory; tak, liczba jest wektorem
 [1]  3  4  5  6  7  8  9 10  4  3  2 -3

> ## 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 = 5, by = 0.4) # liczby od 1 do 5 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

> seq(1, 5, 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

> seq(1, 10, length.out = 20) # R ustala przyrost, aby długość = 20
 [1]  1.00000  1.47368  1.94737  2.42105  2.89474  3.36842  3.84211  4.31579  4.78947  5.26316
[11]  5.73684  6.21053  6.68421  7.15789  7.63158  8.10526  8.57895  9.05263  9.52632 10.00000

> ## 4. 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"
```

Strefa Eksperta 2.2

Wektor możesz utworzyć jeszcze w jeden sposób, wykorzystując funkcję:

```
vector(mode = "logical", length = 0)
```

Tworzy ona wektor o zadanym typie i długości. Do wyboru masz też inne typy: `integer`, `numeric`, `character`. W czym pomocny jest ten nowy sposób? Wyobraź sobie, że z każdym przebiegiem pętli zwiększasz rozmiar wektora o jeden. Ponieważ **R** nie wie, jak długi ostatecznie będzie wektor, to takie dynamiczne zwiększanie bardzo go spowalnia. Z kolei jeżeli wcześnie zarezerwujesz miejsce w pamięci, właśnie przy użyciu tej funkcji, wtedy zdecydowanie skrócisz czas obliczeń.

2.3.2. Operacje na wektorach.

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

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

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

- operatory logiczne: `!`, `|`, `&` (negacja, alternatywa, koniunkcja); alternatywę i koniunkcję dla skalarów przedstawiają odpowiednio symbole: `||`, `&&`; przykładowo:

  `!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
```

- operator binarny `%in%`, który oznacza *należy* i zastępuje matematyczny symbol $\in$; przykład

```
x %in% y
```

Zacznijmy od **operatorów arytmetycznych**. Jeżeli masz wektory takiej samej długości, to działania wykonywane są element po elemencie, np. dla operatora dodawania: $[1,4]+[2,5]=[3,9]$. Na tym mógłbym zakończyć opis działania, gdyby nie to, że R wykona operacje również wtedy, gdy wektory mają różną długość — uważaj na to. Działa to tak, że krótszy z wektorów jest powielany tyle razy, aby zrównał się z długością tego dłuższego. Jest to tzw. ***reguła zawijania*** (*recycling rule*). Przykładowo, chcesz dodać dwa wektory: $[5,7,3]$ i $[1,3,7,4,9,2]$. Pierwszy zostanie powielony dwa razy, więc ostatecznie będzie miał postać: $[5,7,3,5,7,3]$. Dopiero teraz zostanie dodany do drugiego. Nie zostaniesz o tym fakcie poinformowany przez program, bo długość jednego jest wielokrotnością długości drugiego. Zobacz: pierwszy i drugi mają długości 3 i 6 odpowiednio, a 6 jest wielokrotnością 3.

Zapewne zastanawiasz się, co dzieje się w sytuacji, gdy długość większego nie jest wielokrotnością długości mniejszego. Reguła zawijania zostanie zastosowana, ale o tym R poinformuje cię odpowiednim komunikatem. Może spróbujesz dodać jakieś dwa wektory, aby wygenerować taki 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. Pomnożyć liczbę 2 przez wektor $[5,7,3]$, to inaczej pomnożyć dwa wektory: $[2,2,2]$ i $[5,7,3]$ element po elemencie. Jaki będzie wynik? Na końcu będzie 6?

```
> 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 in x + y: 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 elementy wektora y do 2 potęgi
[1]  1 25 81  0  9 25

> z^2 + y # podnieś elementy z do 2 potęgi i dodaj y
[1] 65 86  9 64 84 -5
```

Jeżeli użyjesz **operatora logicznego, relacyjnego lub binarnego** to w wyniku zawsze otrzymasz wektor logiczny z kombinacją stanów: `TRUE`, `FALSE`. Zanim przejdziemy do przykładów, które rozwieją twoje wątpliwości nt. tych operatorów, chciałbym zwrócić twoją uwagę na dwie kwestie. Pierwsza — używaj

operatorów logicznych `|`,`&` tylko do wektorów logicznych. Jeżeli zapiszesz `x|y`, miej pewność, że zarówno `x` jak i `y` są typu logicznego. Druga — operator relacyjny równy (`==`) ma dwa znaki równości. Często, na samym początku przygody z R, jest on zastępowany jednym znakiem równości, co jest źródłem wielu błędów.

```
> ## Rozważmy krótki przykład: dzienny utarg z wizyt
> ## w kolejnych dniach tygodnia. Pracuś?
> stawka <- c(100, 70, 90, 150, 120, 110, 130) # koszt wizyty u specjalisty
> ilePacjent <- c(3, 5, 4, 4, 1, 7, 3) # liczba przyjętych pacjentów
> utarg <- stawka * ilePacjent
> utarg
[1] 300 350 360 600 120 770 390

> utarg != 150 # który utarg jest różny od 150
[1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE

> utarg == 150 # który utarg jest równy 150
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE

> utarg >= 300 # który utarg przynajmniej 300
[1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE

> utarg > 350 | utarg < 200 # który utarg większy od 350 lub mniejszy od 200
[1] FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE

> utarg < 350 & utarg > 250 # który utarg między 250 a 350
[1]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE

> 150 %in% stawka # czy 150 należy do zbioru wartości stawka
[1] TRUE

> c(70, 300) %in% stawka # która wartość wektora należy do stawka
[1]  TRUE FALSE
```

Zauważ, że w niektórych przykładach R wykonuje dwa kroki. Wynikiem pierwszego są dwa wektory logiczne: `utarg > 350` oraz `utarg < 200`. Drugi krok to już operacja z wykorzystaniem logicznego lub (`|`). Użyliśmy go, bo mamy 2 wektory logiczne.

2.3.3. Odwołania do elementów wektora.

Do każdego elementu wektora możesz się odwołać, wskazując indeks (pozycję) tego elementu, który ujmujesz w nawiasy kwadratowe. Tak naprawdę `[` jest tzw. operatorem indeksowania. Musisz wiedzieć, że numerowanie elementów w R zaczyna się od 1 (w niektórych językach od 0, np. w Python). Dla przykładu weźmy zdefiniowany wektor `utarg` z poprzedniego podrozdziału. Jeżeli napiszesz w konsoli: `utarg[5]` to zostanie wzięty 5 element wektora `utarg`, czyli wartość 120.

Bardzo ważna jest umiejętność wyboru elementów, dlatego poniżej zamieszczam kilka możliwości, dla jakiegoś hipotetycznego wektora `x`:

1. `x[liczba]` — wybierz element będący na pozycji `liczba`, np. `liczba=2`, wtedy wybrany zostanie drugi element;
2. `x[-liczba]` — minus oznacza: weź wszystko oprócz elementu o indeksie `liczba`;
3. `x[wektor]` — elementami obiektu `wektor` jest wektor indeksów, np. x[c(1, 8, 3)] oznacza wybranie elementów o indeksach: 1, 8 i 3 (dokładnie w takiej kolejności);
4. `x[-wektor]` — wybierz wszystkie elementy oprócz tych, których indeksy zawiera `wektor`;
5. `x[wektor_logiczny]` — `wektor_logiczny` musi mieć taką samą długość jak wektor `x`; `TRUE` oznacza wzięcie elementu, `FALSE` przeciwnie; jeżeli `x` ma 3 elementy, to np. `x[c(TRUE, TRUE, FALSE)]` zwróci pierwsze dwa elementy i pominie ostatni.

Zobacz, jak te różne strategie wyboru możemy zastosować do wcześniej zdefiniowanego wektora `utarg`.

```
> utarg
[1] 300 350 360 600 120 770 390

> utarg[3] # wybierz 3 elment
[1] 360

> utarg[3:5] # wybierz elementy od 3 do 5, bo 3:5 = c(3, 4, 5)
[1] 360 600 120

> utarg[c(1, 3, 4, 5)] # wybiera elementy o wskazanych indeksach
[1] 300 360 600 120

> utarg[-2] # wybierz wszystkie oprócz elementu na pozycji 2
[1] 300 360 600 120 770 390

> utarg[-c(1, 4)] # wybierz wszystkie bez elementu na pozycji: 1 i 4
[1] 350 360 120 770 390

> doktor <- c(1, 3, 4, 7)
> utarg[doktor] # wybierz elementy o indeksach zapisanych w wektorze: doktor
[1] 300 360 600 390

> utarg[c(TRUE, FALSE, TRUE, TRUE, TRUE, FALSE, FALSE)] # wybierz element tam gdzie TRUE
[1] 300 360 600 120
```

Postaraj się jeszcze przeanalizować poniższe przykłady — celowo nie zamieszczam wyników.

```
> utarg[utarg > 300]
> utarg[utarg > 350 | utarg < 200]
> utarg[utarg < 350 & utarg > 250]
```

Ponieważ wiesz, w jaki sposób odwoływać się do dowolnych elementów wektora, więc zmiana takich elementów jest prosta. Wystarczy, że przypiszesz starym elementom wartości nowe. Załóżmy, że wysokość utargu przekraczająca 400 ma mieć wartość 1000. Dodatkowo element pierwszy i trzeci zamienimy odpowiednio na 0 i 1.

```
> ## Aby nie nadpisywać naszego wektora utarg tworzymy kopię
> utarg2 <- utarg
> utarg2[utarg2 > 400] <- 1000 # krok 1
> utarg2[c(1, 3)] <- c(0, 1) # krok 2
> utarg2
[1]    0  350    1 1000  120 1000  390
```

2.3.4. Wybrane funkcje dla wektorów. Zapewne zgodzisz się, że operatory arytmetyczne dają ograniczone możliwości przetwarzania wektorów. Dlatego jeśli zapytasz, jak policzyć logarytm z elementów wektora, jak znaleźć wartość największą wektora czy jego długość, odpowiem: skorzystaj z wbudowanych funkcji. Poniżej zamieszczam podstawowe funkcje matematyczne, których argumentami mogą być również wektory. W tym wypadku operacje wykonywane są element po elemencie. Poniżej tabeli znajdziesz krótkie przykłady ich wykorzystania.

Tabela 2.2. Funkcje matematyczne dla wektorów

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(x)`	$x! = 1 \cdot 2 \cdot \ldots \cdot x$
`choose(n, k)`	Symbol Newtona $\frac{n!}{k!(n-k)!}$
`abs(x)`	Wartość bezwzględna z 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 N(0,1)
> x
 [1]  0.1320709 -1.0377221  2.1451641 -0.4278806  0.4105311  0.0254474  0.5412996 -0.0197622
 [9] -1.1788083 -2.0211874

> round(x, 1) # zaokrągl do 1 miejsca po przecinku
 [1]  0.1 -1.0  2.1 -0.4  0.4  0.0  0.5  0.0 -1.2 -2.0

> exp(x) # oblicz wartości funkcji e w punktach x
 [1] 1.141189 0.354261 8.543443 0.651889 1.507618 1.025774 1.718238 0.980432 0.307645 0.132498

> abs(x) # oblicz wartości bezwzględne
 [1] 0.1320709 1.0377221 2.1451641 0.4278806 0.4105311 0.0254474 0.5412996 0.0197622 1.1788083
[10] 2.0211874

> log(abs(x)) # najpier oblicz wartość bezwzględną, później log. naturalny
 [1] -2.024416  0.037028  0.763216 -0.848911 -0.890304 -3.671141 -0.613782 -3.923985  0.164504
[10]  0.703685
```

Kontynuujemy przegląd funkcji wbudowanych w R. Teraz poznasz tzw. funkcje agregujące. Czytając opis funkcji spróbuj wyobrazić sobie sytuację, w której mogłyby się przydać. Przykładowo, masz wektor z wartościami 100 transakcji zrealizowanych za pomocą karty. Jaką informację możesz uzyskać, posługując się tymi funkcjami? Następnie zapoznaj się z przykładami, które zamieściłem poniżej.

Tabela 2.3. Funkcje agregujące dla wektorów

Funkcja	Opis
`length(x)`	Długość (liczba elementów) wektora x
`max(x, na.rm = FALSE)`	Największa wartość z x; usunie braki danych, gdy `na.rm = TRUE`
`min(x, na.rm = FALSE)`	Najmniejsza wartość z x; usunie braki danych, gdy `na.rm = TRUE`
`sum(x, na.rm = FALSE)`	Suma wszystkich wartości x; usunie braki danych, gdy `na.rm = TRUE`
`prod(x, na.rm = FALSE)`	Iloczyn wszystkich wartości x; usunie braki danych, gdy `na.rm = TRUE`
`sort(x,decreasing = FALSE)`	Sortuje (rosnąco) wartości x; gdy `TRUE` – malejąco
`sample(x, n, replace = TRUE)`	Losowanie `n` elementów wektora x ze zwracaniem (`replace=TRUE`) lub bez zwracania (`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 pierwszego elementu największego i najmniejszego
`is.na(x)`	Zwraca wektor logiczny, w którym `TRUE` pojawia się tylko wtedy, gdy jest brakująca obserwacja. Jeśli $x = [1, \text{NA}, 5, \text{NA}]$ to operacja `is.na(x)` zwróci `FALSE, TRUE, FALSE, TRUE`
`unique(x)`	Usuwa duplikaty wektora x. Jeśli $x = [1,3,2,1,3,2,1]$ to operacja `unique(x)` zwróci $[1,3,2]$
`range(x)`	Zwraca dwie wartości z x: najmniejszą i największą, jeżeli $x = [1,5,3,2]$ to `range(x)` zwróci $[1,5]$
`table(x)`	Zwraca tabelę kontyngencji. Jeżeli $x = [\text{"}a\text{"}, \text{"}b\text{"}, \text{"}a\text{"}, \text{"}a\text{"}]$ to `table(x)` zwróci $a = 3, b = 1$

```
> ## Przykład: z wektora wartości od 1 do 100 wylosuj 10 liczb
> set.seed(76) # ustaw ziarno generatora (gwarantuje identyczność losowania)
> los <- sample(1:100, 10, replace = FALSE)
> los
 [1]  1 28 50 74 16 87 21 78  6 10

> max(los)
[1] 87

> sum(los)
[1] 371

> sort(los) # argument decreasing pominięty, dlatego użyty domyślny FALSE
 [1]  1  6 10 16 21 28 50 74 78 87
```

```
> zestawienie <- c(range(los), sum(los), length(los))
> zestawienie
[1]   1  87 371  10

> sum(los)/length(los) # oblicz średnią arytmetyczną
[1] 37.1
```

Na szczególną uwagę zasługują funkcje `which()`, które z pewnością przyjdą ci z pomocą wiele razy. Zapamiętaj: funkcja ta zwraca indeksy (pozycje elementów). Jeżeli interesują cię nie tylko indeksy, ale również elementy które odpowiadają tym indeksom, to potrzebujesz dodatkowego kroku. Ale po kolei — spójrz na poniższy przykład.

```
> los
 [1]  1 28 50 74 16 87 21 78  6 10

> which(los > 65) # które elementy są większe od 65, pokaż ich indeksy
[1] 4 6 8

> which.max(los) # pokaż indeks elementu największego (pierwszego napotkanego)
[1] 6
```

Zauważ, że indeksom: 4, 6, 8 odpowiadają wartości 74, 87, 78, które faktycznie są większe od 65. **R** jak już pisałem, zwrócił tylko te indeksy. A jak wyświetlić te wartości? Podejście jest identyczne, jak przy odwoływaniu się do elementów wektora. Musisz w nawiasach kwadratowych umieścić wektor, którego wartościami są numery indeksów czyli:.

```
> los[which(los > 65)]
[1] 74 87 78
```

> **WARTO WIEDZIEĆ**
>
> Kiedyś na pewno staniesz przed pytaniem jak połączyć/skleić dwa wektory. Przykładowo chcesz utworzyć wektor: `klient_1, klient_2,...,klient_100` i zastanawiasz się, czy musisz 100 razy pisać prawie to samo. Nie, bo z pomocą przychodzą dwie funkcje:
>
> ```
> paste(..., sep = " ", collapse = NULL)
> paste0(..., sep = "", collapse = NULL)
> ```
>
> Argumenty `sep` i `collapse` możesz pominąć, ale wtedy ustawione zostaną na widoczne wartości domyślne. Pierwszy odpowiada separacji między każdą parą elementów, drugi odnosi się do separacji między złączonymi już parami. Brzmi zawile? Przeanalizuj poniższy przykład.
>
> ```
> > paste0("klient_", 1:3)
> [1] "klient_1" "klient_2" "klient_3"
>
> > x <- c("biały", "żółty", "niebieski")
> > y <- "kolor"
> > paste(y, x, sep = " CO ", collapse = "|| ")
> [1] "kolor CO biały|| kolor CO żółty|| kolor CO niebieski"
> ```

2.4. Macierze

Na pewno pamiętasz, z kursów matematycznych, czym jest macierz. Przypomnijmy jednak: macierz jest tablicą dwuwymiarową, składającą się z elementów rozmieszczonych w wierszach i kolumnach. Poniższa macierz **X** składa się z n wierszy i k kolumn

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

Widoczne 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. Powyższa macierz jest wymiaru $n \times k$, a więc `dim` jest wektorem dwuelementowym `c(n, k)`.

To podobieństwo do wektora ma swoje konsekwencje: w R macierz składa się z elementów tego samego typu — identycznie jak w wypadku wektora (rozdz. 2.3.1). Podobieństw do wektora jest znacznie więcej, dlatego często będę się odwoływał do tego, co napisałem w wektorach.

2.4.1. Tworzenie macierzy.

Jeśli chcesz utworzyć macierz w R, wykorzystaj funkcję:

```
matrix(wektor, nrow, ncol),
```

w której: `wektor` — jest wektorem, `nrow` — oznacz liczbę wierszy, `ncol` — oznacza liczbę kolumn. Zauważ, że wystarczy podać tylko jeden argument (`nrow` lub `ncol`), gdyż ten brakujący zostanie wyznaczony na podstawie znajomości długości wektora. Weźmy przykład: wektor ma długość 50, a liczba kolumn wynosi 5. Ile mamy wierszy? Oczywiście mamy $50/5 = 10$ wierszy, więc ten argument w funkcji możesz pominąć. Mam jeszcze jedną uwagę: elementy macierzy tworzone są w kolejności kolumnowej — najpierw tworzona jest pierwsza kolumna, później druga itd. Jeśli chcesz utworzyć macierz w kolejności wierszowej, dodaj argument do funkcji: `byrow = TRUE`. Prześledźmy mechanizm tworzenia macierzy na kilku 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; wstaw nrow = 4
     [,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
```

2.4.2* Operacje na macierzach.

Jeżeli wykorzystasz operatory arytmetyczne: dodawanie, odejmowanie, mnożenie i dzielenie, to poznany sposób zachowania się wektorów możesz przenieść na grunt macierzy (rozdz. 2.3.2). Przypomnę tylko, że operacje/działania wykonywane są zgodnie z zasadą: element po elemencie. Przeanalizuj poniższe przykłady.

```
> 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 # nie jest to działanie znane z algebry macierzy
     [,1]    [,2] [,3]
[1,] 10.0 11.6667    3
[2,] 12.5 10.0000   10

> mac1 * mac2 # to też nie jest mnożenie znane z algebry macierzy
     [,1] [,2] [,3]
[1,]   10  105   75
[2,]   50  160  360
```

Oprócz powyższych operacji, możesz również wykonywać działania na macierzach poznane na algebrze, czy innych kursach matematycznych. Iloczyn macierzy wyznaczysz, jeśli użyjesz operatora: `%*%`. Przydatne mogą się też okazać: funkcja transpozycji `t()`, funkcja zwracająca elementy diagonalne macierzy `diag()` czy funkcja zwracającą macierz odwrotną `solve()`. Z kolei wyznacznik macierzy policzysz wykorzystując funkcję `det()`. Przykłady ilustrujące zamieszczam poniżej.

```
> (X <- matrix(round(rnorm(16), 1), nrow=4)) # tworzymy macierz
     [,1] [,2] [,3] [,4]
[1,]  0.2  1.1 -0.8 -0.5
[2,] -0.6  1.5  2.0  0.7
[3,]  0.9  0.0  0.5  1.4
[4,]  0.3 -1.0 -0.1  1.2

> X %*% X # mnożenie
      [,1]  [,2]  [,3]  [,4]
[1,] -1.49  2.37  1.69 -1.05
[2,]  0.99  0.89  4.41  4.99
[3,]  1.05 -0.41 -0.61  1.93
[4,]  0.93 -2.37 -2.41  0.45

> t(X) # transpozycja: zamiana wierszy z kolumnami
     [,1] [,2] [,3] [,4]
[1,]  0.2 -0.6  0.9  0.3
[2,]  1.1  1.5  0.0 -1.0
[3,] -0.8  2.0  0.5 -0.1
[4,] -0.5  0.7  1.4  1.2

> diag(X) # elementy na przekątnej
[1] 0.2 1.5 0.5 1.2

> solve(X) # macierz odwrotna
          [,1]        [,2]        [,3]      [,4]
[1,] -0.453438 -0.56984299  1.28316188 -1.353546
[2,]  0.653763  0.26935571 -0.00676773  0.123173
[3,] -0.832431 -0.00135355  0.48998376 -0.917704
[4,]  0.588793  0.36681104 -0.28559827  1.197888

> X %*% solve(X) # powinniśmy otrzymać macierz jednostkową (ale numeryczna dokładność)
            [,1]         [,2]         [,3]         [,4]
[1,] 1.00000e+00 -2.77556e-17  0.00000e+00 -2.22045e-16
```

```
[2,] 2.22045e-16  1.00000e+00 -1.11022e-16 -2.22045e-16
[3,] 0.00000e+00  0.00000e+00  1.00000e+00  0.00000e+00
[4,] 0.00000e+00  0.00000e+00  0.00000e+00  1.00000e+00
```

2.4.3. Odwołania do elementów macierzy. Do elementów macierzy odwołujesz się, podobnie jak w wypadku wektorów, za pomocą operatora indeksowania `[`. Ponieważ mamy 2 wymiary, więc musisz podać dwa indeksy oddzielone przecinkiem. Pierwszy — odnosi się do numerów wiersza, drugi — wskazuje na numery kolumn. Jeżeli zamiast indeksu zostawisz miejsce puste, poinformujesz R, że ma wziąć wszystko. W rodz. 2.3.3 opisałem kilka sposobów odwołania się do elementów wektora. Z macierzą postępujesz dokładnie identycznie. Jedyna trudność polega na tym, że masz 2 wymiary. Poniższe przykłady wyjaśnią ci istotę wyboru podmacierzy jeszcze lepiej.

```
> 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 # pokaż elementy macierzy
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    8    4    8    2    1    8    5    1    7     4
[2,]    1    4   10    9    1    5    6    9    3     6
[3,]   10   10    2    1    8    5    3    3    6     3
[4,]    3    7    7    7    2    9   10    6    4     6
[5,]   10    6    5    2    2    9    3    6    2     8
[6,]    9    5    6   10    1    5    2    9    1     6
[7,]    9    7    8    4    9    7    7    5    3     2

> x[2, 5] # wybierz wartości: z drugiego wiersza i piątej kolumny
[1] 1

> x[, 5] # wybierz  wartości: ze wszystkich wierszy i piątej kolumny
[1] 1 1 8 2 2 1 9

> x[1, ] # wybierz wartości: z pierwszego wiersza i wszystkich kolumn
 [1] 8 4 8 2 1 8 5 1 7 4

> x[, 5:7] # wybierz wartości: ze wszystkich wierszy i kolumn od 5 do 7
     [,1] [,2] [,3]
[1,]    1    8    5
[2,]    1    5    6
[3,]    8    5    3
[4,]    2    9   10
[5,]    2    9    3
[6,]    1    5    2
[7,]    9    7    7

> x[, c(5, 7)] # wybierz wartości: ze wszystkich wierszy i kolumn 5 i 7
     [,1] [,2]
[1,]    1    5
[2,]    1    6
[3,]    8    3
[4,]    2   10
[5,]    2    3
[6,]    1    2
[7,]    9    7

> x[, -5] # wybierz wszystkie wartości, pomijając kolumnę 5
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
[1,]    8    4    8    2    8    5    1    7    4
[2,]    1    4   10    9    5    6    9    3    6
[3,]   10   10    2    1    5    3    3    6    3
[4,]    3    7    7    7    9   10    6    4    6
[5,]   10    6    5    2    9    3    6    2    8
[6,]    9    5    6   10    5    2    9    1    6
[7,]    9    7    8    4    7    7    5    3    2
```

```
> x[c(2, 6), -c(5, 7)] # wybierz wartości: z wiersza 2 i 6 oraz ze wszystkich kolumny oprócz 5 i 7
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    1    4   10    9    5    9    3    6
[2,]    9    5    6   10    5    9    1    6
```

2.4.4. Wybrane funkcje dla macierzy. Macierze i wektory są podobne, dlatego funkcje które opisałem w rozdz. 2.3.4, możesz zastosować do macierzy. Tę listę rozszerzymy o funkcje dodatkowe — zamieszczam je w poniższej tabeli. Poniżej znajdziesz również przykłady ich użycia.

Tabela 2.4. Funkcje dla macierzy

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 macierz
`rbind(x, y)`	Łączy wierszowo dwie macierze (lub 2 wektory) w jedną macierz
`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 kolumny: `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) # wartość: pierwsza - liczba wierszy, druga - liczba kolumn
[1] 4 5

> ncol(X) # liczba kolumn
[1] 5

> nrow(X) # liczba wierszy
[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)) # Utwórz macierz
     [,1] [,2] [,3]
[1,]   -1   -5   -9
[2,]   -2   -6  -10
[3,]   -3   -7  -11
[4,]   -4   -8  -12

> cbind(X, Y)  # Połącz macierze X i Y kolumnowo
     [,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
```

Czy uważasz, że funkcja `apply()` zasługuje na osobne omówienie? Ja tak uważam, bo w R istnieje wiele podobnych funkcji, które pozwolą ci szybko wykonać operacje, bez użycia pętli. Jeżeli poznasz istotę działania tej funkcji, łatwo zrozumiesz też inne. A więc wyobraź sobie sytuację, w której chcesz wykonać identyczne operacje na każdej kolumnie macierzy (bądź wierszu). Załóżmy, że będzie to

operacja sumowania. Jak mogłyby wyglądać kroki? Prześledź poniższe rozwiązanie.

```
> ## Chcemy zsumować kolumny w macierzy (niech liczba kolumn = 2)
> ## Następnie wyniki sumy przypisać do 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, a wynik przypisujemy do nazwy: sumaKolumn
> sumaKolumn
[1] 10 26
```

Zapewne dostrzegasz nieefektywność takiego podejścia. Słusznie zadajesz pytanie: a co jeśli kolumn mam dużo więcej? Właśnie w takich sytuacjach możesz wspomagać się funkcją `apply()`, która wywołuje inną funkcję (podajesz jaką) operującą na każdym wierszu lub kolumnie macierzy. Jak się później dowiesz, ona działa na każdym obiekcie dwuwymiarowym. Teraz możesz porównać oba rozwiązania. Które wybierzesz?

```
> ## Dla wcześniejszego przykładu
> sumaKolumn <- apply(x, 2, sum) # operacje na kolumnach, bo 2; funkcja to suma
> sumaKolumn
[1] 10 26

> apply(x, 1, sum) # operacje na wierszach bo jest 1
[1]  6  8 10 12
```

Jeszcze jedna uwaga: za funkcję można przyjąć jedną z wbudowanych w R (spróbuj dla `var()`, `mean()`, `median()`) lub napisać własną funkcję. Jak napisać funkcję dowiesz się z rozdziału 3.3.

2.5. Czynniki

W podrozdziale o wektorach napisałem, że mogą one być typu: liczbowego (naturalne, rzeczywiste), logicznego i znakowego. Jest jeszcze jeden typ, trochę podobny do tego ostatniego – **typ czynnikowy** (*factor*). Jego wprowadzenie do struktur danych R jest powiązane z koncepcją zmiennych kategorialnych (nominalne, porządkowe) występujących w statystyce. Jak się przekonasz, ten typ może być naprawdę użyteczny. Czasami jest wręcz niezbędny, jeśli chcesz oszacować parametry modelu (np. modelu regresji) z takiego typu zmiennymi.

2.5.1. Tworzenie czynników. Aby utworzyć obiekt typu czynnikowego, użyj funkcji `factor()`, a za obligatoryjny argument `x` przyjmij wektor (dowolnego typu). Spójrz na trzy warianty:

```
factor(x)
factor(x, ordered = TRUE)
factor(x, levels = ..., labels = ..., ordered = ...)
```

Argumentu opcjonalnego `ordered = TRUE` użyj wtedy, gdy zmienna ma charakter porządkowy. Z kolei `levels` (poziomy) i `labels` (etykiety) będą przydatne, jeżeli wektor `x` jest typu numerycznego. Szczegółowo wyjaśnię to na przykładzie za chwilę. Teraz zapamiętaj tylko, że każdemu poziomowi (każdej liczbie) przyporządkowujesz etykiety (słowny opis), np. 1 ⟶mężczyzna, 2 ⟶kobieta.

Przykład zaczniemy od zdefiniowania wektora typu znakowego, następnie zapytamy o jego typ (`typeof()`) oraz strukturę (`str()`). Po prawej stronie przekształcimy ten wektor na czynnik, a dodatkowo wywołamy funkcję `levels()`, która zwraca poziomy czynnika.

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

> typeof(mieszka) # jaki typ obiekt
[1] "character"

> str(mieszka) # struktura obiektu
 chr [1:4] "miasto" "miasto" "wieś" "miasto"
```

```
> ## Przekształcamy wektor na czynnik
> mieszka <- factor(mieszka)
> mieszka
[1] miasto miasto wieś   miasto
Levels: miasto wieś

> typeof(mieszka)
[1] "integer"

> str(mieszka)
 Factor w/ 2 levels "miasto","wieś": 1 1 2 1

> levels(mieszka) # jakie poziomy
[1] "miasto" "wieś"
```

Nasuwają ci się jakieś wnioski z tej krótkiej analizy? Zbierzmy je. Po pierwsze — zgodnie z naszym zamierzeniem, zmienną typu znakowego `mieszka` przekształciliśmy na czynnik, który ma dwa poziomy (*levels*). Ta informacja pojawia się po wyświetleniu obiektu jak i również jest skutkiem użycia funkcji `levels()`. Po drugie — czynnik reprezentowany jest przez zbiór liczb całkowitych dodatnich 1 i 2. Popatrz na kolejność: jedynce odpowiada miasto, dwójce wieś. Takie przyporządkowanie wynika z porządku alfabetycznego. Po trzecie wreszcie — R zawsze rozpoczyna liczenie od wartości 1.

Strefa Eksperta 2.3

Konwersja czynnika na liczby

Ponieważ czynnik jest zbiorem liczb całkowitych dodatnich z dodatkowym atrybutem `levels`, więc możesz te liczby wyciągnąć, używając np. funkcji `as.numeric()`. Dla przykładu o miejscu zamieszkania mamy

```
> as.numeric(mieszka)
[1] 1 1 2 1
```

Takie podejście nie sprawdzi się, jeżeli masz wektor liczb całkowitych przekształcony na czynnik. Dzieje się tak dlatego, że `as.numeric()` nie zwraca wartości takiego wektora, ale numery użyte w kodowaniu. Problem rozwiążesz używając najpierw funkcji `as.character()`. Przeanalizuj poniższy przykład.

```
> wek <- factor(c(9, 10, 27, 30))
> wek
[1] 9  10 27 30
Levels: 9 10 27 30

> as.numeric(wek) # liczby użyte w kodowaniu
[1] 1 2 3 4

> as.numeric(as.character(wek)) # przekształć na typ znakowy, później numeryczny
[1]  9 10 27 30
```

W kolejnym przykładzie pokażę, jak z wektora numerycznego utworzyć czynniki o zadanych poziomach i etykietach. Najpierw wygenerujmy wektor liczb, którego wartości odwołują się do poziomów wykształcenia.

```
> ## Losujemy 20 liczba ze zbioru: 1, 2, 3, 4
> set.seed(1234) # Ustawienie ziarna generatora liczb (można pominąć)
> (edu <- sample(1:4, 20, replace = TRUE))
 [1] 4 4 2 2 1 4 3 1 1 2 4 4 2 3 2 2 2 3 2 4
```

Następnie zakładamy, że kodowanie przebiega według schematu: 1 — podstawowe, 2 — średnie, 3 — licencjat, 4 — magisterium. Mamy więc stworzyć czynnik, którego poziomami będą opisy poziomu wykształcenia. Postępujemy podobnie jak w wypadku zmiennej `mieszka`, a więc wywołujemy funkcję `factor()`, z dodatkowymi argumentami: `levels` oraz `labels` (poziomy oraz etykiety). Dodatkowo poinformujemy R, że mamy do czynienia ze zmienną porządkową: `ordered = TRUE`.

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

```
+                 ordered = TRUE)
> eduOrd
 [1] magisterium magisterium średnie     średnie     podstawowe  magisterium licencjat
 [8] podstawowe  podstawowe  średnie     magisterium magisterium średnie     licencjat
[15] średnie     średnie     średnie     licencjat   średnie     magisterium
Levels: podstawowe < średnie < licencjat < magisterium
```

2.5.2. Operacje, odwołania i wybrane funkcje dla czynnika. Jak już wiesz, czynniki są wektorami specjalnego przeznaczenia. Dlatego opisane przeze mnie: operacje, odwołania i funkcje mają również zastosowanie do czynników (zob. rozdz. 2.3.1). W tej części chcę zwrócić twoją uwagę ma dwie funkcje. Pierwszą już znasz — to funkcja `table()` podana na str. 17), więc na niej nie będę się skupiał. Jednak zachęcam cię, abyś jej użył do czynnika `eduOrd`. Druga — `droplevels()` — usuwa nieużywane poziomy czynnika. Opiszę ją na przykładzie.

Załóżmy, że chcesz usunąć jakieś poziomy. Przyjmijmy, że jest to poziom: `podstawowe` czynnika `eduOrd`. Zapewne zrobisz to w następujący sposób:

```
> (eduOrd2 <- eduOrd[eduOrd != "podstawowe"]) # usuń wartości: podstawowe
 [1] magisterium magisterium średnie     średnie     magisterium licencjat   średnie
 [8] magisterium magisterium średnie     licencjat   średnie     średnie     średnie
[15] licencjat   średnie     magisterium
Levels: podstawowe < średnie < licencjat < magisterium

> table(eduOrd2) # tabela liczebności zawiera podstawowe
eduOrd2
 podstawowe     średnie   licencjat magisterium
          0           8           3           6
```

Zauważ, że usuwając poziom `podstawowe`, nie usuwasz atrybutu `levels` — dalej są 4 poziomy. Konsekwencje mogą być różne. Przykładowo, jeśli zbudujesz wykres słupkowy, to pojawi się ta kategoria z wartością 0. Zapewne chcesz tego uniknąć i usunąć ten poziom, więc posłużysz się funkcją `droplevels()`.

```
> (eduOrd2 <- droplevels(eduOrd2)) # usuń nieużywane poziomy
 [1] magisterium magisterium średnie     średnie     magisterium licencjat   średnie
 [8] magisterium magisterium średnie     licencjat   średnie     średnie     średnie
[15] licencjat   średnie     magisterium
Levels: średnie < licencjat < magisterium

> table(eduOrd2) # tabela liczebności
eduOrd2
    średnie   licencjat magisterium
          8           3           6
```

2.6. 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**. Dlatego jedna kolumna może składać się z liczb rzeczywistych odnoszących się do liczby ludności, druga zawierać nazwy miejscowości, a trzecia opisywać wykształcenie. Tak zorganizowany zbiór danych znasz np. z arkuszy kalkulacyjnych. Bardzo często wiersze nazywamy przypadkami, albo utożsamiamy z obserwacjami. Kolumny natomiast traktujemy jako zmienne.

2.6.1. Tworzenie ramek danych. W R obiekt o tak opisanej strukturze utworzysz, używając funkcji

```
data.frame(col1, col2, ...)
```

w której każdy `col` jest wektorem kolumnowym o dowolnym typie ale identycznej długości. Jeśli wektory będą różnić się długością, wtedy R zastosuje regułę zawijania, którą już znasz (zob. str. 14).

Utwórzmy, na dwa sposoby, ramkę danych z trzech wektorów.

```
> ## 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(sok, cena, cukier)
> dfSok
        sok cena cukier
1     kubus 1.20   11.5
2     pysio 1.35   12.0
3      leon 1.65   10.0
4 bobo frut 1.99    9.6

> ## Drugi, możliwy sposób
> dfSok2 <- data.frame(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))
> dfSok2
        sok cena cukier
1     kubus 1.20   11.5
2     pysio 1.35   12.0
3      leon 1.65   10.0
4 bobo frut 1.99    9.6
```

2.6.2. Odwołania do elementów ramki danych. Do elementów ramki danych odwołasz się w taki sam sposób, jak do elementów macierzy (zob. str. 21). Zachęcam cię jednak do używania nazw kolumn jako identyfikatora odwołań. Takie podejście jest bezpieczniejsze, jeżeli do ramki danych dodajesz/usuwasz kolejne kolumny lub zmieniasz ich kolejność. Zapoznaj się z poniższymi sposobami odwołań.

- `moja_ramka$nazwa_kolumny` — po nazwie ramki umieszczasz operator $, po którym z kolei podajesz nazwę kolumny; wynikiem jest wektor.
- `moja_ramka[, "nazwa_kolumny"]` — na drugiej pozycji, która odnosi sie do kolumny, podajesz nazwę kolumny/zmiennej ujętej w cudzysłowy; zamiast nazwy kolumny możesz podać wektor nazw.
- `moja_ramka["nazwa_kolumny"]` — ramka danych jest przypadkiem szczególnym listy (zob. rozdz. 2.7), dlatego taki sposób odwołania jest właściwy (dla macierzy nie). Nie jest on równoważny temu powyżej. Zwróć uwagę na wynik tej operacji — pojedynczy `[` zwraca zawsze obiekt tej samej klasy (tutaj ramka danych — *data frame*).
- `moja_ramka[["nazwa_kolumny"]]` — działa podobnie do poprzedniego z tą różnicą, że zwraca wektor (bo jest podwójny nawias).

Prześledź sposób zachowania się każdego z opisanych odwołań, analizując poniższe przykłady. Ramkę danych `dfSok` stworzyliśmy już wcześniej.

```
> ## Wykorzystamy poprzednią ramkę danych: dfSok
> dfSok$cena # równoważnie: dfSok[, 2]
[1] 1.20 1.35 1.65 1.99

> dfSok["cena"] # zwraca ramkę danych, bo pojedyncze [
  cena
1 1.20
2 1.35
3 1.65
4 1.99

> dfSok[["cena"]] # zwraca wektor, bo podwójny [[
[1] 1.20 1.35 1.65 1.99

> dfSok[c("sok", "cukier")] # zwraca ramkę danych. Równoważnie: dfSok[, c(1,3)]
        sok cukier
```

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

Zapamiętaj 2.1

Każda kolumna w ramce danych ma swoją nazwę. Choć możesz się odwoływać poprzez numer kolumny, to zachęcam do używania nazwy. Jeżeli pracujesz z dużą ramką danych i dodajesz lub usuwasz kolumny, zmieniasz ich kolejność, to numery kolumn też ulegają zmianie. Dlatego bezwzględnie zapamiętaj to odwołanie:

`moja_ramka$nazwa_kolumny`

Jeżeli w **R**Studio napiszesz nazwę ramki danych, a po niej umieścisz symbol $, wtedy pojawia się lista rozwijana z nazwą kolumn. Zacznij pisać nazwę lub najedź na właściwą i wciśnij ENTER.

2.6.3. Operacje na ramkach danych.

Na ramkach danych wykonasz podobne (choć nie wszystkie) operacje jak na macierzach i wektorach. Możesz więc dodawać lub usuwać kolumny i wiersze — czym zajmiemy się już teraz. Ale możesz też wybrać jakieś wiersze lub kolumny i utworzyć podzbiór oryginalnego zbioru danych. W praktyce dość często tak robimy. O tym podstawowym sposobie napiszę w kolejnych podrozdziałach.

Wiesz już, jak można dodawać kolumny i wiersze za pomocą funkcji `cbind()` i `rbind()`. W wypadku ramek danych istnieje jeszcze jedna możliwość. Jeżeli chcesz dodać pojedynczą kolumnę, użyj operatora $. Z kolei kolumnę usuniesz przypisując jej wartość `NULL`. Te dwie operacje zilustruję krótkim przykładem: chcemy dodać kolumnę o nazwie `witC` do ramki `dfSok` o wartościach $[24, 20, 18, 32]$ oraz usunąć kolumnę `cukier`.

```
> dfSok$witC <- c(24, 20, 18, 32)
> dfSok$cukier <- NULL
> dfSok
        sok cena witC
1     kubus 1.20   24
2     pysio 1.35   20
3      leon 1.65   18
4 bobo frut 1.99   32
```

W kolejnych 3 podrozdziałach zapoznasz się, na przykładach, z podstawowymi sposobami wyboru przypadków lub wierszy. Pojawią się tam dwa lub nawet trzy sposoby rozwiązania tego samego zadania. Aby **R** wyświetlił tylko ostatni, dla pozostałych utworzę obiekty nazwane `y1`, `y2`. Zapewniam cię, że warto te podejścia znać pomimo tego, że istnieje bardziej przejrzysty i szybszy sposób. Na pewno go polubisz. Ale o nim napiszę później, gdy zapoznamy się z tzw. środowiskiem *tidyverse*. Teraz potrzebujemy zbioru danych do przykładów. Wygenerujemy go przy użyciu poniższego kodu — skopiuj go i wklej do konsoli.

```
## Generujemy wektory do ramki danych
ileObser <- 20
set.seed(12345) # użyj, jeśli chcesz mieć identyczne wartości
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)
```

Wygenerowane wektory danych posłużą nam do stworzenia ramki danych (czy mogę stworzyć macierz z tych wektorów?). Nazwa zbioru danych powinna mówić coś o samych danych. Kieruję się jednak celem dydaktycznym — łatwiej dostrzec różnice przy krótszych nazwach — i nazywam ramkę po prostu `x`.

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

Wybór przypadków

Zwróć uwagę, że słowo przypadek jest synonimem obserwacji i wiersza. Zapoznaj się z poniższymi zadaniami i ich rozwiązaniami. Jeśli czegoś nie rozumiesz, to przeanalizuj przykład rozbijając rozwiązanie na części. Przeanalizujmy pierwszy: `x[x[, 1] == "m", ]`. Zaczynasz od elementów najbardziej wewnętrznych. Co to jest `x[, 1]` — to pierwsza kolumna ramki danych. Co otrzymasz pisząc: `x[, 1] == "m"` — ponieważ sprawdzasz każdy element pierwszej kolumny, pod kątem występowania litery `m`, więc rezultatem będzie wektor logiczny z kombinacją stanów `TRUE`, `FALSE`. Ostatecznie jeżeli taki wektor umieścisz na pierwszym miejscu w nawiasie kwadratowym ramki danych `x[,]`, to zostaną wybrane te wiersze, którym odpowiada wartość `TRUE`, a pominięte te z wartością `FALSE`. Dlaczego wiersze — bo pierwszy indeks odnosi się do wierszy, drugi natomiast do kolumn (analogia do Excela).

```
> x
   plec wiek mieszka papierosy wwwGodziny
1     m   59  miasto         6          6
2     m   58  miasto         1          6
3     m   57    wies         3         14
4     m   49    wies         4          4
5     k   20  miasto         2         13
6     k   31  miasto         2          6
7     k   39    wies         0         11
8     k   27  miasto         6         11
9     m   31    wies         1          9
10    m   22  miasto         6         12
11    k   28  miasto         4         11
12    k   33  miasto         0          7
13    m   32    wies        10          2
14    k   39  miasto         4          4
15    k   35    wies         0         10
16    k   35    wies         0          3
17    k   51    wies         0          8
18    k   53  miasto         0         10
19    k   60    wies         9          0
20    m   51  miasto         4          2
```

Lista przykładów:

1. Wybierz tylko mężczyzn

```
> y1 <- x[x[, 1] == "m", ] # sposób 1 lub: x[x[1] == "m", ]
> y2 <- x[x$plec== "m", ] # sposób 2
> x[x[, "plec"] == "m", ] # sposób 3 lub: x[x["plec"] == "m", ]
   plec wiek mieszka papierosy wwwGodziny
1     m   59  miasto         6          6
2     m   58  miasto         1          6
3     m   57    wies         3         14
4     m   49    wies         4          4
9     m   31    wies         1          9
10    m   22  miasto         6         12
13    m   32    wies        10          2
20    m   51  miasto         4          2
```

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

```
> y1 <- x[x[, 4] > 5, ] # sposób 1
> y2 <- x[x$papierosy > 5, ] # sposób 2
> x[x[, "papierosy"] > 5, ] # sposób 3
   plec wiek mieszka papierosy wwwGodziny
1     m   59  miasto         6          6
8     k   27  miasto         6         11
10    m   22  miasto         6         12
13    m   32    wies        10          2
19    k   60    wies         9          0
```

3. Wyłącz 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
> x[x[, "papierosy"] != 0 & x[, "papierosy"] != 1, ] # sposób 3
```

```
   plec wiek mieszka papierosy wwwGodziny
1     m   59  miasto         6          6
3     m   57    wies         3         14
4     m   49    wies         4          4
5     k   20  miasto         2         13
6     k   31  miasto         2          6
8     k   27  miasto         6         11
10    m   22  miasto         6         12
11    k   28  miasto         4         11
13    m   32    wies        10          2
14    k   39  miasto         4          4
19    k   60    wies         9          0
20    m   51  miasto         4          2
```

4. Wybierz 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
> x[x[, "papierosy"] == 0 & x[, "wwwGodziny"] >= 8, ] # sposób 3
```

```
   plec wiek mieszka papierosy wwwGodziny
7     k   39    wies         0         11
15    k   35    wies         0         10
17    k   51    wies         0          8
18    k   53  miasto         0         10
```

5. Wybierz 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, ] # sposób 2
> x[x[, "plec"] == "k" & x[, "mieszka"] == "wies" & x[, "papierosy"] == 0, ] # sposób 3
```

```
   plec wiek mieszka papierosy wwwGodziny
7     k   39    wies         0         11
15    k   35    wies         0         10
16    k   35    wies         0          3
17    k   51    wies         0          8
```

6. Wyłącz 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
> x[x[, "wiek"] > 50 | x[, "wiek"] < 30, ] # sposób 3
```

```
   plec wiek mieszka papierosy wwwGodziny
1     m   59  miasto         6          6
2     m   58  miasto         1          6
3     m   57    wies         3         14
5     k   20  miasto         2         13
8     k   27  miasto         6         11
10    m   22  miasto         6         12
11    k   28  miasto         4         11
17    k   51    wies         0          8
18    k   53  miasto         0         10
19    k   60    wies         9          0
20    m   51  miasto         4          2
```

Wybór zmiennych

Zapoznając sie z poniższymi przykładami zapewne zauważysz, że odwoływanie poprzez liczby jest krótsze i czasami łatwiejsze (ostatni przykład). Z drugiej strony pisałem, że lepiej wykorzystywać nazwy kolumn — podtrzymuję to. Wspomniane przeze mnie środowisko *tidyverse* ułatwi i znacząco uprości odwoływanie się do nazw.

1. Wybierz zmienne: `wiek, papierosy, wwwGodziny`

```
> y1 <- x[, c(2, 4, 5)] # sposób 1
> y2 <- x[, c("wiek", "papierosy", "wwwGodziny")] # sposób 2
> head(y2, 2) # wyświetl tylko 2 wiersze
```

```
  wiek papierosy wwwGodziny
1   59         6          6
2   58         1          6
```

2. Wybierz wszystkie zmienne: od `wiek` do `papierosy`

```
> y1 <- x[, -c(1, 5)]
> y2 <- x[, c("wiek", "mieszka", "papierosy")]
> head(y2, 2) # wyświetl tylko 2 wiersze
```

```
  wiek mieszka papierosy
1   59  miasto         6
2   58  miasto         1
```

3.★ Wybierz pierwszą zmienną, oraz zmienne od `mieszka` do `wwwGodziny`

```
> y1 <- x[, -2]
> y2 <- x[, !names(x) %in% "wiek"] # ! - zaprzeczamy, że należy
> head(y2, 2) #wyświetl tylko 2 wiersze
```

```
  plec mieszka papierosy wwwGodziny
1    m  miasto         6          6
2    m  miasto         1          6
```

Wybór przypadków i zmiennych

Choć wiesz już tyle, że bez trudu poradzisz sobie z jednoczesnym wyborem przypadków i zmiennych, to dla kompletności rozważań przedstawiam jeszcze jeden przykład.

1. Wybierz zmienne `plec` i `papierosy` oraz te respondentki, które wypalają mniej niż 5 papierosów. Przedstaw rozwiązanie w 2 krokach.

```
> krok1 <- x[, c("plec", "papierosy")] # wybierz zmienne (wyświetl krok1)
> krok2 <- krok1[krok1$plec == "k" & krok1$papierosy < 5, ] # wybierz przypadki z krok1
> krok2
```

```
   plec papierosy
5     k         2
6     k         2
7     k         0
11    k         4
12    k         0
14    k         4
15    k         0
16    k         0
17    k         0
18    k         0
```

2. Powyższe zadanie rozwiąż w pojedynczym kroku

```
> jedenKrok <- x[x$plec == "k" & x$papierosy < 5, c("plec", "papierosy")]
```

2.6.4. Wybrane funkcje dla ramek danych. Pisząc o macierzach, w tabeli podałem listę funkcji (str. 22). Możesz je również zastosować do ramek danych. Poniżej przedstawiam kolejne, które w szczególności polecane są dla ramek danych.

Tabela 2.5. Funkcje dla ramek danych

Funkcja	Opis
`names(x)`	Zwraca nazwy kolumn ramki danych x
`na.omit(x)`	Usuwa te wiersze ramki danych x, w których występują braki danych.
`head(x, 6)`	Pokazuje 6 (domyślnie) pierwszych wierszy ramki danych, co jest równoważne zapisowi: `head(x)`. Możesz zmienić liczbę 6 na inną.
`str(x)`	Pokazuje strukturę ramki danych x: wymiar, listę zmiennych, typy zmiennych oraz kilka początkowych wartości
`sapply(x, fun)`	Wykonuje operację na każdej kolumnie definiowaną przez funkcję fun. Bardzo podobne działanie jak `apply(x, 2, fun)`

Funkcja `names()` nie tylko pozwala nam wyświetlić nazwy kolumn, ale również jest bardzo pomocna w zmianie nazw już istniejących. Poniżej znajdziesz dwa przykłady takich zmian. W pierwszym zmienimy wszystkie nazwy, w drugim tylko jedną. Zwróć uwagę na sposób tworzenia ramki danych — nie podajemy nazw kolumn, więc R sam ją ustala,

```
> ## Tworzymy ramkę danych
> dfZwierz <- data.frame(c(1, 4, 0), c("tak", "tak", "nie"), c("kot", "pies", "tygrys"))
> names(dfZwierz) # tak wyglądają nazwy
[1] "c.1..4..0."                "c..tak....tak....nie.."     "c..kot....pies....tygrys.."
```

Teraz zmieniamy nazwy. Ponieważ nazwy tworzą wektor, więc do jego elementów odwołujesz się w poznany już sposób.

```
> names(dfZwierz) <- c("ile", "ma", "zwierzak") # zmieniamy wszystkie paskudne nazwy
> dfZwierz
  ile  ma zwierzak
1   1 tak      kot
2   4 tak     pies
3   0 nie   tygrys

> names(dfZwierz)[c(1, 3)] <- c("ile_zwierz", "zwierz_4nogi") # zmieniamy ponownie: tylko 1 i 3 nazwę
> dfZwierz
  ile_zwierz  ma zwierz_4nogi
1          1 tak          kot
2          4 tak         pies
3          0 nie       tygrys
```

Zapamiętaj 2.2

Podgląd ramki danych

Funkcji `head()` oraz `str()` użyliśmy już wcześniej, więc znasz ich działanie. Muszę jednak dodać, że jeśli chcesz zerknąć na ramkę danych, to właśnie te funkcje ci w tym pomogą. Gdy mamy dużą ramkę danych (bardzo wiele wierszy i kolumn), nie ma sensu wpisywanie w konsoli nazwy ramki danych i wyświetlanie jej zawartości. R i tak wyświetli tylko niewielką część i najprawdopodobniej nie zobaczysz nazw wierszy.

Strefa Eksperta 2.4

W obiektach dwuwymiarowych podobnych do macierzy (a więc i ramki danych) nazwy kolumn i wierszy możesz definiować lub, jeśli są już zdefiniowane, zwracać ich nazwy za pomocą funkcji: `colnames()` i `rownames()`. Jeśli masz macierz z nazwami kolumn, to możesz ich użyć do odwołania. Pamiętaj, aby wykorzystać nawiasy kwadratowe wewnątrz których musi być przecinek.

```
> (macSok <- as.matrix(dfSok)) # Uzgadanie typów: wszystko typu znakowego
     sok         cena   witC
[1,] "kubus"     "1.20" "24"
[2,] "pysio"     "1.35" "20"
[3,] "leon"      "1.65" "18"
[4,] "bobo frut" "1.99" "32"

> colnames(macSok)
[1] "sok"  "cena" "witC"

> macSok[, "cena"]
[1] "1.20" "1.35" "1.65" "1.99"

> rownames(macSok) # brak nazw wierszy
NULL

> rownames(macSok)  <- LETTERS[1:4] # dafiniujemy nazwy
> macSok
  sok         cena   witC
A "kubus"     "1.20" "24"
B "pysio"     "1.35" "20"
C "leon"      "1.65" "18"
D "bobo frut" "1.99" "32"

> macSok["B", ]
    sok    cena    witC
"pysio"  "1.35"    "20"
```

Mam ostatnią uwagę: `names()` nie używaj do macierzy tylko do ramek danych.

2.7.★Listy

Ten rozdział możesz całkowicie pominąć, a tę możliwość sygnalizuje gwiazdka. Zdecydowałem się jednak listy opisać, choć to będzie bardzo zwięzłe, wręcz telegraficzne ujęcie. Listy bardzo często towarzyszą analitykowi danych i bez ich znajomości nasza wiedza na temat struktur danych będzie niepełna.

Listę mogą tworzyć elementy dowolnego typu. Z tego względu uważamy ją za najbardziej złożoną strukturę w R. Często wykorzystujemy ją do przechowywania różnego typu danych czy informacji. Przykładowo, weryfikujemy hipotezę statystyczną testem t-Studenta wywołując odpowiednią funkcję (`t.test()`). 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.

2.7.1. Tworzenie listy.

Listę utworzysz wywołując funkcję `list()`. Ogólnie możemy przyjąć, że

```
list(obiekt1, obiekt2, ...)
```

utworzy listę, w której każdy `obiekt` może być wektorem, macierzą, ramką danych czy nawet listą. Jeśli każdy obiekt jest wektorem takiej samej długości, to mamy strukturę odpowiadającą ramce danych. Wniosek nasuwa się taki, że ramka danych jest przypadkiem szczególnym listy. Możesz się o tym przekonać pytając R czy ramka danych `mojaRamka` jest listą: `is.list(mojaRamka)`. Stwórzmy naszą pierwszą listę.

```
> ## 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?).

2.7.2. Odwołania do elementów listy. Do elementów listy odwołasz się wykorzystując poznane już operatory: `[` lub `[[`. Pierwszy z nich, o czym wspominałem wcześniej, zwraca element tego samego typu co obiekt główny — czyli listę. W wyniku użycia drugiego operatora otrzymasz obiekty, które tworzą listę. W wypadku utworzonej listy `mojaLista` otrzymasz odpowiednio: wektor, wektor, wektor, macierz. Prześledźmy to na przykładzie.

```
> #W Wybór elementów z listy
> mojaLista[2] # weź drugi obiekt listy; wynikiem jest zawsze lista bo [
[[1]]
[1] 20 10 15 16

> mojaLista[[2]] # weź drugi obiekt listy; wynikiem jest wektor bo [[
[1] 20 10 15 16

> mojaLista[c(2,3)] # weź drugi i trzeci obiekt listy
[[1]]
[1] 20 10 15 16

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

> # weź drugi obiekt listy a następnie 3 element tego obiektu; równoważne z mojaLista[[2]][[3]]
> mojaLista[[c(2, 3)]]
[1] 15
```

Do elementów listy możesz odwoływać się wykorzystując nazwy. Odbywa się to w identyczny sposób jak w wypadku ramek danych, a więc z wykorzystaniem operatora `$`. Nadajmy nazwy poszczególnym obiektom listy, bo przecież ich nie mają.

```
> 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 obiektu
[1] "Ewa"   "Nella" "Tammy"

> mojaLista[c("imie", "cena")] # wybór dwóch obiektów listy
$imie
[1] "Ewa"   "Nella" "Tammy"

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

2.7.3. Operacje na listach. Listy możesz również modyfikować, dodając bądź usuwając elementy. Do tych celów najlepiej użyć operatora `[[`. Oczywiście możesz też użyć pojedynczego nawiasu kwadratowego, ale pamiętaj, że przypisany obiekt musi być typu lista. Wyjaśniam to na poniższym przykładzie.

```
> mojaLista[["ostatni"]] <- c("dodany" , "elemen", "listy")
> mojaLista
$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

$ostatni
[1] "dodany" "elemen" "listy"

> # Równoważnie do powyższego
> mojaLista["ostatni"] <- list(c("dodany" , "elemen", "listy"))
> mojaLista["ostatni"] <- NULL # A teraz usuwamy ten obiekt
```

2.7.4. Działania na listach — funkcja `lapply()` i `sapply()`. Opisana na stronie 22 funkcja `apply()` pozwalała na automatyzację operacji na wszystkich wierszach bądź kolumnach. W podobny sposób działa funkcja `lapply()` na elementach listy. Pierwszym argumentem jest obiekt typu lista, drugim natomiast jest nazwa funkcji (bądź też tzw. funkcja anonimowa zob. 3.3.2).

W poprzednim przykładzie stworzyliśmy listę o nazwie: `mojaLista`. Policzmy długość jego drugiego obiektu

```
> length(mojaLista[[2]])
[1] 4
```

Gdy chcemy tę procedurę powtórzyć wielokrotnie, dla wszystkich obiektów listy, wtedy użyjemy tej funkcji.

```
> lapply(mojaLista, length)
$tytul
[1] 1

$cena
[1] 4

$imie
[1] 3

$mojamac
```

```
[1] 10
```

Zapamiętaj, funkcja (`lapply()`) zawsze zwraca obiekt typu lista. Jeśli wiemy, że wynikiem może być jakaś prostsza struktura, np. macierz, wektor, wtedy możemy użyć funkcji `sapply()`. Pierwsza litera s w tej funkcji oznacza uproszczenie (*simplified*). A wiec w naszym przykładzie otrzymaliśmy listę o 4 obiektach, a każdy ma jeden element. Czy nie możemy uprościć tej struktury — masz jakieś podejrzenia? Sprawdźmy.

```
> sapply(mojaLista, length) # oblicz dlugość (length) każdago elementu listy
  tytul    cena    imie mojamac
      1       4       3      10

> sapply(mojaLista, class) # jakiej klasy są elementy listy
$tytul
[1] "character"

$cena
[1] "numeric"

$imie
[1] "character"

$mojamac
[1] "matrix" "array"
```

Zauważ, że funkcje `length()` i `class()` wykonują takie same operacje na każdym obiekcie listy. Dlatego musisz wybrać takie funkcje, których działanie ma sens dla wszystkich obiektów. W powyższym przykładzie nie użyjemy funkcji `mean()`, która wylicza średnią. I jeszcze jedna uwaga: o ile w trybie interaktywnym funkcja `sapply()` jest użyteczna, to nie polecam jej używać we własnych programach czy tworzonych pakietach. Może być źródłem błędów, gdyż zwraca obiekty różnego typu i czasem trudno ten typ przewidzieć.

2.8.★ Funkcje R w rachunku prawdopodobieństwa

Wiele funkcji rozkładów prawdopodobieństwa znajduje się w pakiecie `stats`, który wczytywany jest zawsze podczas uruchamiania R. Sięgnij do pomocy (wpisz: `?Distributions`), aby zapoznać się z pełną listą. W tym rozdziale opiszę 4 rozkład ciągłe: normalny, t-Studenta, jednostajny, wykładniczy oraz 2 rozkłady dyskretne: dwumianowy i Poissona. Ten materiał nie jest obowiązkowy, ale jeżeli chcesz zobaczyć, jak żmudne rachunki czy szukanie wartości w tablicach można zastąpić jedną linijką kodu, to zachęcam cię do zapoznania się z poniższym tekstem.

W ramach każdego rozkładu wyróżniamy 4 funkcje: gęstość lub rozkład prawdopodobieństwa (`d` — *density*), dystrybuantę (`p` — *probability*), kwantyle (`q` — *quantile*), generator liczb pseudolosowych (`r` — *random*). Przedrostek który widzisz w nawiasie (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 podasz, wtedy przyjmowane są wartości domyślne widoczne powyżej w definicji funkcji. Argumentami na pierwszej pozycji są: `x` — punkt w którym chcemy obliczyć wartość funkcji gęstości (`d`) i wartość dystrybuanty (`p`), `p` — prawdopodobieństwo (rząd kwantyla), `n` — ile wygenerować obserwacji.

Działanie pierwszych trzech funkcji zilustruję przykładem. Generowanie liczb losowych pozostawiam ci 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 sd = sqrt(b)
> 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
```

Ponieważ funkcje w R są zazwyczaj zwektoryzowane, dlatego argumentami mogą być także wektory. Zobacz jak szybko można obliczyć prawdopodobieństwa dla kilku wartości.

```
> pnorm(c(-1, -0.5, 2.1, 3.5), mean = 0.5, sd = 3) # Oblicz dyst. w punktach
[1] 0.308538 0.369441 0.703099 0.841345
```

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

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

Rozkład jednostajny: `min` i `max` — odpowiednio dolny i górny kraniec przedziału; widoczne wartości są 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. Opis funkcji które przedstawiłem, a dotyczący przedrostków, ma również zastosowanie do rozkładów dyskretnych. Pamiętaj 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 dla `size, prob`

```
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) # Obliczamy Pr(X=7),  a 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 -  to jest nasza odpowiedź
[1] 0.39199
```

Możemy to zadanie rozwiązać, wykorzystując dystrybuantę: $F(x) = \mathbb{P}(X \leqslant x)$. W zadaniu mamy obliczyć:

$$\mathbb{P}(X \geqslant 7) = \mathbb{P}(X > 6) = 1 - \mathbb{P}(X \leqslant 6) = 1 - F(6)$$

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)
```

2.9. Zadania

Obsługa R

Zad. 1. Wykorzystaj odpowiednie funkcje i utwórz wektory, nadając im nazwy (wymyślone). Wektory mają składać się z następujących elementów:

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) "czy", "to", "jest", "wektor z NA"
e) $4, 7, 9, 4, 7, 9, 4, 7, 9, 4, 7, 9, 4, 7, 9, 4, 7, 9.$

Następnie dla każdego podaj: długość (liczba elementów), typ, element najmniejszy i największy. Wartości ostatniego wektora posortuj. Skorzystaj z odpowiednich funkcji.

Zad. 2. Wykorzystaj poniższy skrypt do wygenerowania wektora `cena` w PLN.

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

Następnie zaokrąglij cenę do dwóch miejsc po przecinku. Zdefiniuj nowy wektor, którego wartości będą ceną wyrażoną w EURO; przyjmij kurs wymiany na poziomie 4.28 PLN/EUR. Nowy wektor zaokrąglij do liczb całkowitych, a następnie:

a) znajdź jego wartość największą i najmniejszą;
b) podaj liczbę jego unikalnych elementów, później je posortuj i wyświetl w konsoli R;
c) wykorzystaj wzory i oblicz: sumę elementów ($\sum_{i=1}^{n} x_i$), średnią arytmetyczną ($\frac{1}{n}\sum_{i=1}^{n} x_i$) i geometryczną ($\sqrt[n]{x_1 \cdot x_2 \cdot \ldots x_n}$). Pamiętasz zależność między pierwiastkowaniem a potęgowaniem?
d) podaj liczbę wartości: (1) większych od 13 EUR, (2) mniejszych od 15 EUR i większych od 10 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, skopiuj go 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
```

Utwórz ramkę danych o nazwie `kredyt`, której kolumnami będą następujące wektory: `rataKapitalu`, `odsetki`, `rataKredytu`, `zadluzenie`. Użyj funkcji `class()` by sprawdzić, czy utworzony obiekt faktycznie jest ramką danych. Następnie wykorzystaj odpowiednie funkcje i:

a) wyświetl pierwszych 10 wierszy;
b) pokaż strukturę ramki;
c) jaki jest wymiar ramki danych;
d) wyświetl w konsoli wiersze: (a) od 100 do 125 (b) pierwszych 20 (c) ostatnich 30 (d) od 20 do 30 i od 50 do 60 (e) co dziesiąty wiersz (10, 20, 30 itd.)
e) oblicz sumaryczną wielkość zapłaconych odsetek, rat kredytu i rat kapitałowych.
f)★ od którego okresu wysokość raty kapitałowej (`rataKapitalu`) zaczyna przewyższać wysokość spłacanych odsetek (`odsetki`)? Może przydać 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))
```

a) Zmień domyślne nazwy kolumn na: `masa` i `wysokosc`.
b) 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. Poniżej efekt końcowy (wyświetlam tylko 4 wiersze)
c)★ 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`. Wskazówka: jeżeli `x <- c(30,10,25)` i napiszemy `x > 20` to otrzymamy ciąg: `TRUE, FALSE, TRUE`. Jeżeli zapiszemy `(x > 20)+ 1` to otrzymamy: `2,1,2`. Pamiętasz o rzutowaniu typów?
d) 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:

```
> 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
```

e) Dodaj wiersz do ramki danych, który będzie odnosił się do jakiejś wymyślonej osoby.
f) Wykorzystując operatory `[, [[, $` wybierz różnymi sposobami: kolumnę `masa` a następnie dowolne 2 kolumny.
g) Wybierz te wiersze dla których `BMI` jest większe od 23.
h) Które osoby (nr wiersza) mają wagę prawidłową?

Zad. 5. Umieść w katalogu swojego projektu plik z danymi `satysfakcja.dat` z badań sondażowych i uruchom poniższą linikję:

```
saty <- read.table("satysfakcja.dat", header = TRUE, sep = "\t")
```

W tym zadaniu utworzysz kilka podzbiorów ramki danych `saty`. Zapisz te podzbiory pod wymyślonymi nazwami. Jeśli masz wątpliwości wróć do rozdz. 2.6.3.

a) Wybierz te zmienne, które odnoszą się do edukacji.
b) Wybierz osoby, które wierzą w życie po śmierci i są spod znaku lwa.
c) Wybierz te osoby, których znak zodiaku zaczyna się od litery B.
d) Weź wszystkie zmienne od `wieku` do `płci` włącznie, a następnie z tego zbioru wybierz osoby, które uczyły się więcej niż 19 lat.
e) W zbiorze danych są zmienne, które mają charakter porządkowy (np. częstotliwość czytania gazet). Choć są reprezentowane w postaci czynników, to jednak nie uwzględniono tam charakteru porządkowego. Popraw to.
f) wybierz wszystkie zmienne typu numerycznego (użyj funkcji `sapply()`). Dla nich policz sumę i znajdź wartości największe i najmniejsze.
g) Wybierz kilka zmiennych (o różnym typie) i wykonaj na nich operacje wykorzystując funkcje: `sum()`, `table()`, `unique()`. Pamiętaj: nie wszystkie funkcje można wykorzystać do wszystkich zmiennych.

Zad. 6.* Uruchom poniższą linijkę tworzącą obiekt `reg` typu lista.

```
reg <- lm(y ~ x, data = data.frame(y=rnorm(100), x=rnorm(100) ))
```

Z ilu elementów składa się lista. Jak nazywają się elementy listy. Oblicz długość wszystkich elementów listy oraz powiedz, jakiego są typu. Wydobądź element listy o nazwie `coefficients`.

* *Prawdopodobieństwo*

Zad. 7. Dla zestandaryzowanego rozkładu normalnego i rozkładu t-studenta o 15 stopniach swobody
a) wyznacz kwantyle rzędu $p = 0.85$, $p = 0.99$, $p = 0.27$.
b) oblicz prawdopodobieństwa: $\Pr(X > 1.8)$, $\Pr(X \geqslant 2.47)$.
Do obliczeń wykorzystaj odpowiednie funkcje R.

Zad. 8. 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 jakie jest prawdopodobieństwo tego, że:
a) dokładnie trzy wnioski zostaną odrzucone;
b) 10 wniosków zostanie przyjętych;
c) przynajmniej 10 wniosków zostanie przyjętych.
Dla przypomnienia: rozkład dwumianowy ma postać: $\Pr(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$, $k = 0, 1, \ldots, n$

Zad. 9. 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 $\lambda = 5$. Oblicz 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. 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 warunek jest prawdziwy, wykona następna linijkę — jest ona pomijana, gdy wyrażenie jest fałszywe. Zamieszczam krótki przykład.

```
> if (2==3)    # Jeżeli 2 jest równe 3 to
+     a <- 10  # za a podstaw 10
> exists("a")  # czy obiekt a istnieje
[1] FALSE
```

Jeśli chcemy, aby instrukcji do wykonania było więcej, wtedy musimy ująć je w nawiasy klamrowe. Wszystkie wyrażenia między tymi nawiasami R podda ewaluacji. Zobacz jak wygląda taka składnia 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 {
  wykonajA_jesli_FALSE
  wykonajB_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` będą wykonywane. Przeanalizujmy poniższe przykłady i zobaczmy 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 <- "x musi być numeryczny"
+    }
> srednia
[1] "x musi być numeryczny"
```

Funkcją `is.numeric()` sprawdzamy, czy wektor `x` jest typu numerycznego, bo tylko dla takiego typu (choć dla logicznego również) ma sens liczenie średniej. Warunek jest spełniony (wariant a), więc instrukcje w nawiasie klamrowym są przetwarzane przez R — średnia została policzona. Wariant b) jest bardziej elastyczny, gdyż uwzględnia sytuację, w której podany wektor jest typu znakowego. Jak

wiesz, policzenie średniej w tej sytuacji nie ma sensu i warto o tym użytkownika poinformować. W ramach ćwiczenia spróbuj napisać program, który sprawdza, czy kwota przekracza średnią krajową. W zależności od tej kwoty powinien pojawić się komunikat: `za mało` lub `za dużo`.

W instrukcji `if...else` mamy `warunek`, który zwraca pojedynczą wartość logiczną (`TRUE` lub `FALSE`). Czasami zdarza się, że chcemy poddać testowi każdy element wektora — wtedy mamy ciąg stanów prawda i fałsz, np. `c(1, 6, 3)> 4`. Chociaż możemy użyć pętli (zob. rozdz. 3.2) i sekwencyjnie sprawdzić element po elemencie, to wygodniej jest posługiwać się funkcją:

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

Jej działanie wyjaśniam tym przykładem:

```
> x <- c(1,5,4,3,2,7,8,9,2,4)
> x > 7
 [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE

> ifelse(x > 7, "wieksza", "mniejsza")
 [1] "mniejsza" "mniejsza" "mniejsza" "mniejsza" "mniejsza" "mniejsza" "wieksza"  "wieksza"
 [9] "mniejsza" "mniejsza"
```

Zauważ, że pierwszy argument funkcji jest wektorem wartości logicznych. R sprawdza kolejno, czy `1 > 7`, czy `5 > 7`, czy `4 > 7` itd. W zależności od wyniku przyjmowana jest jedna z wartości: `wieksza` (gdy `TRUE`) lub `mniejsza` (gdy `FALSE`). Musisz wiedzieć, że niezależnie od wartości wektora logicznego R 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, -3)
> ifelse(x >= 0, sqrt(x), sqrt(abs(x)))

Warning in sqrt(x): NaNs produced
[1] 3.00000 2.00000 1.73205
```

Ostrzeżenie pojawia się, gdyż `sqrt(x)` oraz `sqrt(abs(x))` obliczane są dla wszystkich elementów (nawet dla −3). I nie mają znaczenia wartości wektora logicznego. Chociaż wywołanie funkcji gwarantuje, że R wyświetli nam odpowiednie wartości, zależne od wartości wektora logicznego. Inaczej zachowuje się zestaw instrukcji `if...else`. Obliczenia wykonywane są albo dla `TRUE` albo dla `FALSE`. Jeśli byłoby inaczej, wtedy w wariancie b) otrzymalibyśmy komunikat o błędzie. Zapamiętaj te różnice.

3.2. Pętla for

Nieraz musimy wykonać pewną operację wiele razy. Przykładowo stoisz przed zadaniem zbudowania wykresów słupkowych, które pokazują strukturę wykształcenia w każdym województwie. W konsekwencji musisz zbudować 16 prawie identycznych wykresów. Dodatkowo wykresy powinny być zapisane do pliku graficznego `png`. Oczywiście możesz napisać kod dla jednego województwa, a następnie skopiować go 15 razy i trochę zmodyfikować. A co jeśli zamiast 16 województw masz 50 czy 100 największych miast? Właśnie w takich sytuacjach znakomicie sprawdzi się pętla `for`, którą omówię.

Pętla umożliwia wielokrotne wykonywanie poleceń będących w zasięgu pętli. Składnia wygląda następująco:

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

Powyższy zapis czytamy: dla każdego `i` należącego (`in`) do `wektor` wykonaj wszystkie polecenia mieszczące się w nawiasie klamrowym. Każdy przebieg pętli, związany ze zmianą `i` będziemy nazywać iteracją. Czyli w pierwszej iteracji `i` przyjmuje wartość równą pierwszemu elementowi `wektor`, w drugiej iteracji `i` ma wartość równą drugiemu elementowi `wektor` itd. Możesz wybrać dowolną nazwę — nie musi to być `i`.

Poniższy przykład zamieszczam jako ilustrację zachowania pętli. Na pewno do rozwiązania takiego problemu jej nie potrzebujemy.

```
> # Każdy element wektora podnieś do kwadratu i wyświetl wartość
> liczby <- c(5,3,4,-7)
> for (k in liczby) {
+     print(k^2)
+ }
[1] 25
[1] 9
[1] 16
[1] 49
```

Strefa Eksperta 3.1

Pętle a szybkość

Jeżeli zależy ci na szybkości działania programu, to unikaj pętli, które nie są mocną stroną programu. Jednak bardzo często pętle można zastąpić działaniami na wektorach i macierzach. Szybkość takich operacji jest na poziomie języków kompilowanych tj. C++. Do tych operacji **R** wykorzystuje zoptymalizowaną bibliotekę BLAS (*Basic Linear Algebra System*). Jeszcze lepiej zoptymalizowana jest biblioteka `Intel MKL`, która domyślnie jest wykorzystywana w innej (również darmowej) dystrybucji **R** tj. Microsoft R Open.

Na zakończenie tej części napiszemy krótki program, który będzie obliczał sumę w każdej kolumnie macierzy. Czy pamiętasz, że funkcja `apply()` pozwala nam to zadanie rozwiązać szybciej, bez pętli?

```
> # 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 -0.341123
[10]  1.991356

> apply(x, 2, sum) # zamiast pętli
 [1]  0.526071 -0.676366 -0.731426  1.510806 -0.691149 -2.314322  0.143808 -2.189465 -0.341123
[10]  1.991356
```

Przeanalizujmy powyższy program. 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`, sumujemy jej wartości, a powstały wynik zapisujemy jako pierwszy element wektora `suma_x`. Tę procedurę powtarzamy 10 razy.

3.3. Funkcje

Do tej pory poznaliśmy wiele wbudowanych funkcji. Co ciekawe, nawet operatory, np. `[`, są funkcjami, choć sposób ich wywołania jest odmienny. Nadrzędnym powodem tworzenia funkcji jest unikanie powielania tych samych fragmentów kodu. Czy będziemy za każdym razem, kiedy chcemy obliczyć średnią z wektora obserwacji, wykonywali następujące kroki: sumowanie wartości, obliczanie długości wektora, dzielenie sumy przez długość wektora? Na pewno nie — w tej sytuacji wykorzystamy wbudowaną funkcję `mean()`, która te kroki realizuje. Zaobserwuj u siebie pewne zachowanie: czy pisząc kod, nie kopiujesz pewnych jego fragmentów, a następnie wklejasz gdzie indziej i poddajesz drobnym modyfikacjom. Przyjmuje się, że jeśli takie zachowanie pojawiło się już 3 razy, to czas napisać funkcję.

Do ilustracji posłużmy się następującym zadaniem: mamy ramkę danych `ram` o kilku kolumnach nazwanych `kol1, kol2,...` itd. Każdą zmienną w kolumnie standaryzujemy: od wartości zmiennej odejmujemy średnią, a następnie wynik dzielimy przez odchylenie standardowe. Dla dwóch pierwszych kolumn, kod wygląda następująco:

```
> standKol1 <- (ram$kol1 - mean(ram$kol1)) / sd(ram$kol1)
> standKol2 <- (ram$kol2 - mean(ram$kol2)) / sd(ram$kol2)
```

Co zrobiłem: napisałem pierwszą linijkę, którą następnie skopiowałem i wkleiłem poniżej. Zmieniłem tylko nazwę kolumny na `kol2`. Takie podejście jest problematyczne z trzech powodów. Po pierwsze — jeśli liczba kolumn jest duża np. 100, wtedy musiałbym 100 razy to skopiować. Po drugie — istnieją duże szanse pomyłki związanej ze zmianą nazwy kolumn. Po trzecie wreszcie — jakakolwiek modyfikacja skopiowanych fragmentów jest czasochłonna. Aby tych problemów uniknąć powinniśmy napisać funkcję, która dla dowolnego wektora wejściowego, zwraca wektor zestandaryzowany. Zanim taką funkcję napiszemy, przejdźmy do niezbędnej teorii.

3.3.1. Tworzenie funkcji.

Poniżej przedstawiam składnię deklaracji funkcji z dwoma argumentami. Liczba argumentów zależy od nas — może ich nie być w ogóle. Widzisz tam dwa warianty, różniące się funkcją `return()`. W większości wypadków możesz uważać je za równoważne. Czyli funkcja zwraca `jakisObiekt` w obu sytuacjach. Jeśli chcesz dowiedzieć się, kiedy te podejście nie są równoważne czytaj ramkę WARTO WIEDZIEĆ.

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

```
NazwaFunkcji <- function(arg1, arg2){
        cialoFunkcji
        jakisObiekt
      }
```

Funkcję możemy wywołać na dwa, równoważne sposoby.

```
        NazwaFunkcji(arg1 = podaj1, arg2 = podaj2)
    lub
        NazwaFunkcji(podaj1, podaj2)
```

Argumentami funkcji są jakieś obiekty R. Jeśli obiekty zapiszemy w takiej samej kolejności (drugi sposób) co argumenty w definicji funkcji, to nazwy argumentów możemy pominąć. Kolejności nie musimy przestrzegać, jeśli stosujemy pierwszy sposób. Wywołanie: `NazwaFunkcji(arg2 = podaj2, arg1 = podaj1)` jest również poprawne, chociaż z praktycznego punktu widzenia nie znajduję dla niego uzasadnienia.

W momencie wywołania funkcji 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. Jeśli wewnątrz funkcji napiszemy `x <- 1`, a następnie spróbujemy wyświetlić wartość `x` w konsoli, wtedy R zgłosi błąd, że nie ma takiego obiektu. Jeśli wewnątrz funkcji operujemy na obiekcie, np. `y`, to musi on być wewnątrz tej funkcji zdefiniowany (np. `y <- 2*x`) lub być argumentem funkcji. Jeśli nie jest, wtedy R szuka tego obiektu poziom wyżej, czyli w tzw. środowisku globalnym. Zachowanie to zilustruję poniższym przykładem (lewa strona).

```
> y <- 2
> Dodaj <- function(x) {
+     x + y
+ }
> Dodaj(5)
[1] 7
```

```
> Dodaj <- function(x, y) {
+     x + y
+ }
> Dodaj(5, 2)
[1] 7
```

Wywołujemy funkcję `Dodaj` z jednym argumentem `x = 5`. Ponieważ w ciele funkcji mamy działanie `x + y`, wiec R szuka `y`. Nie było go w argumencie jak i w ciele funkcji, więc bierze go ze środowiska globalnego (poza funkcją). Oczywiście, jeżeli i tam go nie znajdzie to zgłosi błąd. Pisanie funkcji w ten sposób jest bardzo złą praktyką, gdyż może być powodem wielu problemów czy błędów — nigdy tego nie rób. A jak napisać funkcję poprawnie? Zobacz kod z prawej strony.

Powiedzmy jeszcze o pewnym wariancie. Załóżmy, że najczęściej wywołujesz funkcję `Dodaj` z wartością `y = 2`. Choć czasami podstawiasz też inną wartość. Wtedy najlepszym rozwiązaniem jest ustawienie tego argumentu na wartość domyślną. W takiej sytuacji R weźmie tę wartość tylko wtedy, gdy jej nie podasz. Przeanalizuj poniższy przykład.

```
> Dodaj <- function(x, y = 2) {
+     z <- x + y # można, ale nie trzeba przypisywać wynik obiektowi z
```

```
+     z
+ }
> Dodaj(5) # bez y, wtedy domyślnie y = 2
[1] 7

> Dodaj(5, 10) # jest y, którego wartość zmieniamy na 10
[1] 15
```

> **WARTO WIEDZIEĆ**
>
> Po wywołaniu funkcja zwraca wartość obiektu, np. wartość z w powyższym przykładzie. Zapamiętaj, że funkcja może zwrócić tylko jeden obiekt, a obiektem może być cokolwiek np. wektor, ramka danych, lista itp. Jeśli do zwrócenia obiektu wykorzystamy funkcję `return()`, wtedy dodatkowym efektem, oprócz zwrócenia wartości, będzie zatrzymanie działania funkcji na linijce, w której pojawiła się `return()`. Zobacz przykład:
>
> ```
> > Dodaj <- function(x, y) {
> +     z <- x + y
> +     return(z) # zwraca i zatrzymuje się
> +     x - y
> + }
> > Dodaj(5, 10)
> [1] 15
> ```
>
> ```
> > Dodaj <- function(x, y) {
> +     z <- x + y
> +     z
> +     x - y # to zwróci, a nie z
> + }
> > Dodaj(5, 10)
> [1] -5
> ```

Stwórzmy jeszcze funkcję, która oblicza średnią i wariancję z podanego wektora liczbowego. Utrudnimy sobie zadanie i nie wykorzystamy wbudowanych funkcji, które liczą te dwie statystyki — użyjemy wzorów. Zakładamy, że wektor wejściowy x jest typu numerycznego i nie zawiera braków danych.

```
> SredWar <- function(x) {
+    # Oblicz średnią i wariancję z wartości wektora
+    # Zwróć wektor o2 elementach: srednia i wariancja

+    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)
> SredWar(myVec)
[1] 4.42857 6.81633

> SredWar(rnorm(20)) #średnia z wygenerowanych liczb
[1] 0.286412 1.061766
```

Musisz mieć świadomość, że tak zbudowana funkcja ma pewne ograniczenia — nie jest odporna, np. na wywołanie z takimi argumentami: `SredWar(c("a", "b"))`. Jeżeli piszesz funkcję dla siebie, to zapewne nie musisz tym się martwić — wiesz, że tego robić nie można. Pisząc jednak funkcje z której będą korzystali inni będzie wymagało od ciebie tzw. obsługi błędów.

Chciałbym podzielić się z tobą refleksją na temat tworzenia nazw funkcji. Kiedyś przyjąłem konwencję, że nazwa każdej funkcji zaczyna się dużą literą. Pozostałe obiekty, które tworzę, zawsze zaczynają się małą. Ta strategia pozwala mi na szybkie odróżnienie moich funkcji. Jeśli pracujesz przy dużym projekcie, to liczba tworzonych funkcji może być nawet dwucyfrowa. Osądź, czy to się sprawdzi w twoim wypadku.

Strefa Eksperta 3.2

Obsługa błędów

Wzorcowo napisana funkcja powinna sprawdzać argumenty pod kątem ich sensowności i możliwości wykonania funkcji. Do tego celu wykorzystujemy jedną z dwóch funkcji: `stopifnot()` lub `stop()`. Druga nie tylko zatrzymuje działanie funkcji, ale informuje nas o źródle błędu.

```
stopifnot(warunek_pracy)
```

```
if (warunek_zatrzymania) {
  stop("Nasz komentarz", call. = FALSE)
}
```

W wypadku funkcji `SredWar` sprawdzenie może mieć następującą postać. A może rozbudujesz je o obsługę braków danych.

```
if (!is.numeric(x) || length(x) < 2)
    stop("Wektor musi być numeryczny i mieć przynajmniej 2 wartości", call. = FALSE)
```

3.3.2. Funkcje anonimowe. Funkcje anonimowe są funkcjami, które nie mają nazw. Tworzymy je wtedy, gdy potrzebujemy własnej funkcji, której użyjemy tylko raz. Zazwyczaj są to funkcje, które z kolei są argumentem innych funkcji. Dobrym przykładem jest już poznana przez ciebie funkcja: `apply()` (zob. str. 22). Trzecim jej argumentem jest funkcja. Przykładowo jeśli chcielibyśmy policzyć sumę w każdej kolumnie ramki danych `ramka`, wtedy wystarczy napisać: `apply(ramka, 2, sum)`. A co jeśli nie ma takiej wbudowanej funkcji, bo nasza operacja jest niestandardowa? Załóżmy, że chcemy podnieść każdy element w kolumnie do kwadratu, a następnie wynik zsumować — i tak dla każdej kolumny. Wtedy wystarczy taką funkcję anonimową zdefiniować samemu, co przedstawiam poniżej.

```
> # Generujemy wartości dla każdej kolumny
> ramka <- data.frame(kol1 = rnorm(20), kol2 = rnorm(20), kol3 = rnorm(20))
> head(ramka, 2)
       kol1        kol2      kol3
1 -0.254224  0.00311932 0.3986721
2  1.519432 -0.53122362 0.0267003

> apply(ramka, 2, function(x)sum(x^2)) # funkcja anonimowa z argumentem x
   kol1    kol2    kol3
23.8927 25.9290 19.7961
```

Oczywiście możesz najpierw zdefiniować funkcję, która wykonuje takie operacje, a następnie za trzeci argument podstawić jej nazwę. Będzie to miało większe uzasadnienie, jeśli przynajmniej raz ją wykorzystasz.

Od wersji R 4.1 możemy używać skrótowego zapisu przy tworzeniu funkcji anonimowych. Zobacz:

```
> apply(ramka, 2, \(x)sum(x^2)) # function(x) zastąpione \(x)
   kol1    kol2    kol3
23.8927 25.9290 19.7961
```

Strefa Eksperta 3.3

Jest jeszcze jeden sposób tworzenia funkcji anonimowych, jeżeli wykorzystujemy narzędzia *tidyverse*. Poznasz je w rozdz. 5. Dla naszego przykładu

```
> library(tidyverse)
> map_df(ramka, ~sum(.x^2))
# A tibble: 1 x 3
   kol1  kol2  kol3
  <dbl> <dbl> <dbl>
1  23.9  25.9  19.8
```

Składnia funkcji anonimowej zaczyna się od tyldy ~, a jej argumentem jest x z kropką. Wymaga jednak użycia funkcji z *tidyverse*, dlatego użyłem `map_df()`.

3.4. Zadania

Zad. 1. Wykorzystując pętlę oblicz średnią w każdej kolumnie ramki danych `car`.

```
> car <- mtcars[, 1:7]
> head(car, 2)
```

```
              mpg cyl disp  hp drat    wt  qsec
Mazda RX4      21   6  160 110  3.9 2.620 16.46
Mazda RX4 Wag  21   6  160 110  3.9 2.875 17.02
```

Wynik zapisz w wektorze o nazwie `srednia`. Sprawdź poprawność używając funkcji `apply()`.

Zad. 2. Uruchom poniższy fragment generujący oceny:

```
> oceny <- data.frame(ocenaNum = 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). Następnie rozszerz przykład na sytuację, w której elementami wektora będzie słowna nazwa oceny, np. gdy 5 to `b.dobry`. Te dwa wektory dodaj do ramki danych `oceny`.

Zad. 3. Napisz funkcję `WspolRozklad`, która dla wektora numerycznego o minimalnej długości 4, zwróci wartości dwóch charakterystyk rozkładu: współczynnik asymetrii (skośność) i współczynnik spłaszczenia (kurtoza). Do obliczeń wykorzystaj poniższe wzory:

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

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

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

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

Zad. 4. Zmienną która nie ma rozkładu normalnego, możemy poddać tzw. przekształceniu Boxa-Coxa. Mając wartość oryginalnej zmiennej Y oraz wartość parametru $\lambda \neq 0$ stosujemy następującą transformację:

$$\text{sign}(Y) \times \frac{|Y|^{\lambda} - 1}{\lambda}$$

a) Napisz funkcję, która dla wektora wejściowego y i parametru λ zwróci wartość przekształconej zmiennej. Funkcje wartości bezwzględnej i signum są następujące: `abs()`, `sign()`.
b) Napisaną powyżej funkcję zmodyfikuj tak, aby dla wartości ujemnych y i wartości ujemnych λ funkcja zwracała `NA`.
c) W ostatniej modyfikacji funkcji uwzględnij przypadek, w którym $\lambda = 0$ i wtedy transformacja wygląda następująco: $\log(y)$.

ROZDZIAŁ 4 Wczytywanie i zapisywanie danych

Zazwyczaj każdą pracę z danymi rozpoczynamy od wczytania ich do R. Dlatego ten wstępny etap musi zakończyć się pełnym sukcesem, abyśmy mogli przystąpić do właściwej analizy. Zapewne wiesz, że dane mogą być przechowywane w plikach zewnętrznych o różnych formatach. Przykładem mogą być pliki tekstowe o następujących rozszerzeniach: `txt`, `dat`, `csv`. W tym rozdziale poznasz funkcje, które pozwalają zaimportować dane z takich plików tekstowych. Dodatkowo podpowiem, jak wczytać pliki Excela w formacie `xls` i `xlsx`. Następnie nauczysz się procedury odwrotnej — jak dane zapisać do pliku. Zanim jednak przejdziemy do wczytywania i zapisywania danych, muszę poruszyć bardzo ważną kwestię lokalizacji pliku i ścieżki dostępu.

Aby zapisać lub odczytać plik, R musi znać ścieżkę dostępu — gdzie na dysku znajduje się plik. Gdy takiej lokalizacji nie podamy, R stosuje domyślną lokalizację. Jeżeli pracujemy z programem RStudio w systemie projektów (zob. rozdz. 1.2), tą domyślną lokalizacją jest katalog projektu. Wewnątrz funkcji importującej dane podajemy tylko nazwę pliku z rozszerzeniem (a nie całą ścieżkę dostępu), bo R szuka tego pliku wewnątrz naszego projektu. Jedyne co musisz zrobić, to umieścić plik w tym katalogu.

Zapamiętaj 4.1

Lokalizacja katalogu roboczego

Kiedy uruchamiasz RStudio, jeden z katalogów na dysku jest katalogiem roboczym. Na przykład w tym katalogu R szuka plików do importu, zapisuje historię itp. Już wiesz, że pracując w systemie projektów tym katalogiem roboczym jest katalog projektu. Możesz to potwierdzić, wywołując funkcję `getwd()`, która jest skrótem od: *get working directory* (weź katalog roboczy). Jeżeli nie masz projektu, czego nie polecam, to w systemie Windows będzie to zapewne katalog *Moje dokumenty*.

Aby sprawdzić listę plików i katalogów w katalogu roboczym użyj funkcji `dir()`. Czy jest tam importowany plik?

Zapewne na samym początku przygody z R będziesz zapisywać wszystkie pliki bezpośrednio w katalogu roboczym. Z doświadczenia mogę powiedzieć, że najlepszym sposobem organizacji pracy jest zapisywanie zbiorów danych, plików graficznych (wykresy), raportów itp. w osobnych katalogach. Dlatego tę zasadę zastosujemy do zbiorów danych. Od tej pory wszystkie dane będziemy przechowywać w katalogu o nazwie `dane`. Utwórz go wewnątrz katalogu projektu i umieść tam pliki do importu. Decydując się na taki krok musimy powiedzieć R, że pliki znajdują się w tym katalogu. To oznacza, że wewnątrz funkcji importującej napiszemy np.: `"dane/mojplik.txt"` zamiast samej nazwy pliku: `"mojplik.txt"`.

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

4.1. Wczytywanie danych

4.1.1. Wczytywanie danych z plików tekstowych. Funkcja `read.table()` pozwala nam wczytać dane do R, które mają strukturę tabelaryczną — łatwo możemy wskazać wiersze i kolumny. Funkcja ma bardzo rozbudowaną listę argumentów, co sprawia, że radzi sobie z różnymi formatami. Jeżeli podasz tylko jeden argument tej funkcji, którym musi być nazwa pliku, to pozostałe będą miały wartości domyślne. Poniżej przedstawiam składnię funkcji `read.table()`, z kilkoma wybranymi, domyślnymi wartościami argumentów. Wybrałem tylko te, które najczęściej są zmieniane. Więcej możesz przeczytać w pomocy, wpisując w konsoli: `?read.table`.

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

`header`	—	pierwszy wiersz zawiera nazwy zmiennych: tak – `TRUE`, nie – `FALSE`;
`sep`	—	znak separacji kolumn; domyślne `""` oznacza dowolny biały znak; wpisując `"\t"` informujemy o znaku tabulacji;
`dec`	—	znak separacji części całkowitych i dziesiętnych; jeśli liczby są postaci 2,35 wtedy `dec = ","`;
`na.strings`	—	sposób oznaczenia braków danych; domyślnie `NA`;
`encoding`	—	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`.

Chciałbym, aby nie było wątpliwości, więc napiszę jeszcze raz. 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 `FALSE`.

Jeśli mamy pliki tekstowe w formacie `csv`, wtedy dla własnej wygody posłużymy się nakładkami na funkcję bazową `read.table()` tj.: `read.csv2()` i `read.csv()`. Zobacz, jak wygląda wczytanie pliku z wykorzystaniem funkcji bazowej — musimy zmienić domyślne wartości argumentów. Porównaj to z alternatywnym i krótszym zapisem, w którym wykorzystujemy nakładki.

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

```
## read.csv()
read.csv("mojPlik.csv")
read.table("mojPlik.csv", header = TRUE, sep = ",")
```

W katalogu `dane` są m.in. pliki: `powiaty.txt`, `P081.dat`, `zatrudnienie.dat`, `zatrudnienie0.csv`. Wykorzystaj poznane funkcje, aby wczytać ich zawartość. Pamiętaj, aby wynik przypisać do nazwy. Jeśli tego nie zrobisz, to zawartość pliku zostanie wyświetlona w konsoli R — zapewne tylko część, jeśli jest wiele obserwacji. Tego nie chcemy. Proponuję zacząć od wyświetlenia zawartości pliku w programie `Notepad++`. Dzięki temu zobaczysz, czy pierwszy wiersz ma nazwy zmiennych, jaki jest znak separacji kolumn itd. Zastanów się nad każdym argumentem. W pliku `zatrudnienie.dat` braki danych zostały oznaczone liczbą: −1.

Poniżej znajdziesz rozwiązanie. Skorzystaj z niego tylko po zakończeniu powyższego ćwiczenia. Nawet jeśli napotkasz trudności, postaraj się je rozwiązać samodzielnie. W rozwiązaniu uwzględniłem niezbędną liczbę argumentów. Jeśli nie zmieniłbym któregoś, przyjmując tym samym wartości domyślne, nie zaimportowałbym danych poprawnie. Dla przykładu, jeżeli pominąłbym `na.strings=-1`, wtedy −1 byłoby traktowane jako liczba, a nie jako zakodowany brak danych. Zwróć jeszcze uwagę na to, że katalog w którym znajdują się wczytywane poniżej pliki to *dane*. Musisz go utworzyć, a następnie 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")
```

> **WARTO WIEDZIEĆ**
>
> W pakiecie `readr`, wchodzącym w skład grupy pakietów *tidyverse* (zob. rozdz. 5), dostępne są funkcje: `read_csv()` i `read_csv2()`. Czas wczytywania danych przez te funkcje jest nieporównywalnie krótszy od czasu jakiego potrzebują wbudowane funkcje: `read.csv2()` i `read.csv()`. Średnio są one 10 razy szybsze. Oczywiście tę różnicę odczujemy tym bardziej, im rozmiar importowanego pliku będzie większy. Przykładowo, wczytywałem plik o rozmiarze 185 MB, który miał milion wierszy. Funkcja `read_csv2()` potrzebowała niecałych 5 sekund, natomiast jej wbudowanemu odpowiednikowi zajęło to 26 sekund. Zmiana kodowania dla tych funkcji wygląda trochę inaczej, dlatego poniżej zamieszczam przykład z kodowaniem `UTF-8`.
>
> ```
> zatrud1 <- read_csv2("dane/zatrudnienie0.csv", locale = locale(encoding = "UTF-8"))
> ```

4.1.2. Wczytywanie danych z formatów `xlsx` i `xls`. Popularność plików w formacie *xls, xlsx* (domyślne formaty plików MS Excela) sprawia, że chcielibyśmy dane w tych formatach wczytywać do R. Istnieje wiele pakietów, jak np. rewelacyjny pakiet `XLConnect`, pozwalających na taki import. Niektóre z nich są zależne od zewnętrznych programów jak np. wspomniany pakiet od Javy. Wobec tego dobrym wyborem jest pakiet `readxl`, który tworzy grupę pakietów *tidyverse* (zob. rozdz. 5). Składnia z minimalną liczbą argumentów wygląda następująco.

```
read_xls("nazwa_pliku.xls", sheet = NULL) # sheet to numer lub nazwa skoroszytu Excela
read_xlsx("nazwa_pliku.xlsx", sheet = NULL)
```

4.2. Zapisywanie danych do pliku

Ramki danych możemy zapisać do plików tekstowych, używając ogólnej funkcji `write.table()`. Poniżej zamieszczam jej składnię, podobnie jak dla funkcji wczytującej, ograniczając się do kilku argumentów.

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

`rd`	—	nazwa ramki danych, którą chcemy zapisać;
`file`	—	ujęta w cudzysłowy nazwa pliku wraz z rozszerzeniem;
`quote`	—	zmienne typu znakowego są ujęte w cudzysłowy (`TRUE`) albo nie (`FALSE`);
`sep`	—	separatorem kolumn jest spacja; jeśli `quote = FALSE`, wtedy za separator lepiej przyjąć coś innego np. tabulator (`"\t"`);
`na`	—	ujęte w cudzysłowy znaki kodujące braki danych;
`row.names`	—	nazwy (numery) wierszy są zapisane (`TRUE`) albo nie (`FALSE`); warto ustawić na `FALSE`.
`col.names`	—	nazwy kolumn są zapisane (`TRUE`) albo nie (`FALSE`);

Jeśli chcemy zapisać plik do formatu `csv`, również możemy skorzystać z wygodnych nakładek na funkcję bazową: `write.csv2()` i `write.csv()`. Którą wybrać? Jeżeli pracujesz z polskojęzyczną wersją systemu operacyjnego, to zapewne lepszym wyborem będzie pierwsza funkcja. Poniższe przykłady pozwolą ci zapoznać się z różnymi możliwościami zapisu. Użyjemy do tego celu wygenerowanych danych i utworzonej na ich podstawie ramki danych.

```
set.seed(123) # ustawienie ziarna generatora liczb pseudolosowych
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)
dfbad <- data.frame(opinia, wykszt,  wynagro)
head(dfbad, 5)
            opinia              wykszt wynagro
1 Ani tak, ani nie zawodowe lub niższe 3562.99
2 Ani tak, ani nie              wyższe 2627.56
3       Raczej nie zawodowe lub niższe 3976.97
4       Raczej nie              wyższe 2625.42
5 Ani tak, ani nie             średnie 4052.71
```

Zapiszmy ramkę danych `dfbad` do pliku tekstowego na 6 różnych 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 = FALSE)
> write.csv2(dfbad, "dane/badanie4.csv", row.names = FALSE)
> write.csv2(dfbad, "dane/badanie5.csv", row.names = FALSE, quote = FALSE)
> write.csv(dfbad, "dane/badanie6.csv", row.names = FALSE)
```

Pliki pojawią się na dysku. Możesz je otworzyć w `Notepad++` i zobaczyć efekty. Dla ułatwienia analizy, zamieszczam zrzuty do porównania.

badanie1.txt

```
"opinia" "wykszt" "wynagro"
"1" "Raczej nie" "średnie" 2289.59
"2" "Raczej tak" "zawodowe lub niższe" 3256
"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
```

badanie6.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
```

ROZDZIAŁ 5 Przekształcanie i analiza danych z wykorzystaniem narzędzi *tidyverse*

Narzędzia *tidyverse* tworzą pakiety, w których zaimplementowano filozofię podejścia do analizy danych. W wypadku wykresów i pakietu `ggplot2`, jego twórca Hadley Wickham wykorzystał gramatykę grafiki Wilkinsona (12). Narzędzia są odpowiedzią na zapotrzebowanie tych osób, które chcą relatywnie szybko i łatwo przygotować dane, poddać je analizie i ostatecznie wyniki wizualizować. Nie oznacza to, że bez tych pakietów takich operacji nie jesteśmy w stanie zrobić. Jednak redukują one złożoność całej procedury, przez co składnia jest bardziej przejrzysta i intuicyjna.

Zainstaluj zbiorczy pakiet `tidyverse`, aby w jednym kroku zainstalować 30 pakietów. Następnie uruchom poniższe linijki kodu.

```
> library(tidyverse) # Wczytuje tylko wybrane pakiety z tidyverse
> tidyverse_packages() # Pokazuje wszystkie zainstalowane tidyverse
 [1] "broom"         "cli"           "crayon"        "dbplyr"        "dplyr"
 [6] "dtplyr"        "forcats"       "googledrive"   "googlesheets4" "ggplot2"
[11] "haven"         "hms"           "httr"          "jsonlite"      "lubridate"
[16] "magrittr"      "modelr"        "pillar"        "purrr"         "readr"
[21] "readxl"        "reprex"        "rlang"         "rstudioapi"    "rvest"
[26] "stringr"       "tibble"        "tidyr"         "xml2"          "tidyverse"
```

Pierwsza linijka ładuje kilka najczęściej wykorzystywanych pakietów. Przeglądając ich listę możesz stwierdzić, że np. tylko jeden lub dwa pakiety cię interesują. W takiej sytuacji wczytaj je bezpośrednio — w znany ci sposób, np. `library(ggplot2)`. W drugiej linii używam funkcji, która zwraca listę wszystkich zainstalowanych pakietów *tidyverse*. Jeżeli chcesz użyć pakietu, który nie został wczytany pierwszą linią, musisz go wczytać bezpośrednio.

W tym rozdziale będziemy posługiwali się danymi `bankFull.csv`, które zostały zebrane w wyniku przeprowadzenia, przez pewien bank portugalski, kampanii marketingu bezpośredniego. Dane pochodzą z *UCI machine learning repository* (1). Wczytaj dane i zapisz je w obiekcie o nazwie `bank`. Zamieszczam tabelę z oryginalnym opisem zmiennych. Tłumaczeniem zajmę się bezpośrednio w tekście.

Tabela 5.1. Zmienne zbioru danych bank

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. Pięć podstawowych funkcji pakietu `dplyr`

Do podstawowych operacji jakie wykonujemy na zbiorze danych zaliczamy: wybór wierszy lub kolumn, sortowanie wierszy, tworzenie nowych zmiennych na bazie już istniejących, agregację zmiennych z wykorzystaniem statystyk opisowych. Te wszystkie operacje możesz wykonać, używając jedynie 5 funkcji z pakietu `dplyr`. Dodałbym jeszcze jedną, która pozwala na wykonanie tych przekształceń w podziale na grupy, np. poziomy wykształcenia. Poniżej zamieszczam tabelę, w której składnia odnosi się do jakiejś hipotetycznej ramki danych `rd` i zmiennych `zm1, zm2, ...`.

Tabela 5.2. Podstawowe funkcje z pakietu `dplyr`

Funkcja	Opis
`select(rd, zm1, zm5)`	Wybiera z ramki danych `rd` zmienne `zm1` i `zm5`. Zapisując `zm1:zm5` wybieramy te zmienne oraz wszystkie będące między nimi. Dając minus przed nazwą, wykluczamy ją.
`filter(rd, war_1, war_2)`	Wybiera te wiersze, które jednocześnie spełniają warunki: `war_1` i `war_2`. Przecinek pełni rolę operatora logicznego: `&`. Możemy użyć operatora lub: `|`.
`mutate(rd, nowaZm = ...)`	Dodaje zmienną o nazwie `nowaZm`. W miejsce kropek wstawiamy wyrażenie tworzące zmienną, np. `zm1/2`.
`arrange(rd, zm2, desc(zm4))`	Sortuje wiersze w porządku rosnącym dla zmiennej `zm2` i malejącym dla zmiennej `zm4`.
`summarise(rd, statOpis = fun(zm3))`	Oblicza wartość statystyki opisowej `fun` dla zmiennej `zm3` i zapisuje ją pod nazwą `statOpis`, np. aby obliczyć średnią: `mean(zm3)`
`group_by(rd, zm3, zm6)`	Dzieli ramkę danych na grupy, jakie tworzą kombinacje poziomów 2 zmiennych kategorialnych: `zm3`, `zm6`. Jeśli pierwsza ma 2 poziomy, a druga 3, to mamy 6 grup.

Zapamiętaj, że funkcję `group_by()` stosujesz do zmiennych kategorialnych. Możesz też wykorzystać ją do zmiennych ilościowych dyskretnych (licznikowych), jeżeli liczba unikalnych wartości jest nieduża. Przykładem jest zmienna mówiąca o liczbie dzieci czy taż kart kredytowych w gospodarstwie domowym. Na pewno nie użyjesz tej funkcji do zmiennej wynagrodzenie lub stopa bezrobocia.

5.2. Wybór przypadków i zmiennych do analizy

Znasz sposób wyboru przypadków (obserwacji/wierszy) lub zmiennych, bo omówiłem go w rozdziale 2.6.2 i 2.6.3. Wykorzystywaliśmy wtedy dwa operatory: `[` i `$`. Taki sam efekt osiągniemy, jeżeli użyjemy funkcji `filter()` i `select()`. Ich składnie zamieściłem w tabeli 5.2, więc przejdę do przykładów ilustrujących ich zachowanie. Dla porównania wykorzystamy tą samą ramkę danych, co we wspomnianych rozdziałach.

```
> ## Ramka danych
> x <- data.frame(plec, wiek, mieszka, papierosy, wwwGodziny)
> head(x)
  plec wiek mieszka papierosy wwwGodziny
1    m   59  miasto         6          6
2    m   58  miasto         1          6
3    m   57    wies         3         14
4    m   49    wies         4          4
5    k   20  miasto         2         13
6    k   31  miasto         2          6
```

> **WARTO WIEDZIEĆ**
>
> Funkcja `select()` daje nam więcej możliwości wyboru zmiennych.

```
select(rd, plec, wiek) # tylko dwie zmienne
select(rd, -wiek) # wszystkie zmienne oprócz: wiek
select(rd, plec:wiek) # od płci do wieku (w kolejnosci)
select(rd, -(plec:wiek)) # wszystkie oprócz zm: od płci do wieku
select(rd, contains("pyt")) # nazwy zawierajace pyt
select(rd, starts_with("pyt_")) # zmienne zaczynajace sie od pyt_
select(rd, ends_with("_r2019")) # zmienne konczace sie na _r2019
select(rd, num_range("pyt", 1:4)) # od pyt1 do pyt4 (bez kolejnosci)
select(rd, matches("...")) # pasujace do wyrazenia regularnego
select(rd, wiek, everything()) # zmiana kolejności zmiennych: najpierw wiek później pozostałe
select(rd, all_of(wek)) # wszystkie zmienne o nazwach w wektorze wek
select(rd, any_of(wek)) # jak all_of(), ale nie zwraca błędu, gdy zm. nie istnieją
select(rd, where(fun)) # zmienne, dla których fun zwraca TRUE, np. fun = is.numeric
```

Zanim uruchomisz poniższe kody pamiętaj, aby wczytać odpowiedni pakiet. Zwróć również uwagę na brak cudzysłowów, gdy używamy nazw zmiennych.

1. Wybierz niepalące kobiety ze wsi

```
> filter(x, plec == "k", mieszka == "wies", papierosy == 0)
```

```
  plec wiek mieszka papierosy wwwGodziny
1    k   39    wies         0         11
2    k   35    wies         0         10
3    k   35    wies         0          3
4    k   51    wies         0          8
```

2. Wyłącz osoby miedzy 30 a 50 rokiem życia, a następnie weź tylko mężczyzn

```
> filter(x, wiek > 50 | wiek < 30, plec == "m") # (wiek > 50 | wiek < 30) & plec == "m"
```

```
  plec wiek mieszka papierosy wwwGodziny
1    m   59  miasto         6          6
2    m   58  miasto         1          6
3    m   57    wies         3         14
4    m   22  miasto         6         12
5    m   51  miasto         4          2
```

3. Wybierz zmienne: `wiek, papierosy, wwwGodziny`

```
> y <- select(x, wiek, papierosy, wwwGodziny)
> head(y, 2) # wyświetl tylko 2 wiersze
```

```
  wiek papierosy wwwGodziny
1   59         6          6
2   58         1          6
```

4. Wybierz wszystkie zmienne: od `wiek` do `papierosy`

```
> y <- select(x, wiek:papierosy)
> head(y, 2)
```

```
  wiek mieszka papierosy
1   59  miasto         6
2   58  miasto         1
```

5. Wybierz pierwszą zmienną, oraz zmienne od `mieszka` do `wwwGodziny`

```
> y <- select(x, plec, mieszka:wwwGodziny)
> head(y, 2)
```

```
  plec mieszka papierosy wwwGodziny
1    m  miasto         6          6
2    m  miasto         1          6
```

6. Wybierz zmienne `plec` i `papierosy` oraz te respondentki, które wypalają mniej niż 5 papierosów.

```
> krok1 <- select(x, plec, papierosy) # wybierz najpierw zmienne (zobacz jak wygląda krok1)
> krok2 <- filter(krok1, plec == "k", papierosy < 5) # później przypadki
> krok2
```

```
  plec papierosy
1    k         2
2    k         2
```

```
3     k         0
4     k         4
5     k         0
6     k         4
7     k         0
8     k         0
9     k         0
10    k         0
```

Zapewne docenisz czytelność kodu jak i łatwość z jaką udało nam się wykonać powyższe zadania. Dlatego kontynuując ten aspekt wprowadzę pojęcie **operatora potoku**, który sprawdza się wszędzie tam, gdzie analiza wymaga wielu kroków.

W ostatnim przykładzie użyliśmy funkcji `select()`, by następnie wynik tej operacji poddać przekształceniu — czyli działaniu funkcji `filter()`. Oczywiście nic nie stoi na przeszkodzie, abyśmy w jednym kroku wykonali te dwie operacje, co pokazuję poniżej.

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

Takie podejście, na tzw. „cebulkę", już staje się mało czytelne. Jeżeli kroków będzie więcej, to takiej strategii powinniśmy zdecydowanie unikać. Wobec tego masz dwie możliwości. Pierwsza — przypisujesz krokom pośrednim nazwy, podobnie jak to robiliśmy wcześniej. Druga — wykorzystujesz operator potoku. Jest on bardzo ważny, dlatego szczegóły zamieszczam w ramce.

Zapamiętaj 5.1

Operator potoku `%>%`

Operator do przetwarzania potokowego `%>%` przekazuje wszystko co znajduje się po jego lewej stronie jako pierwszy argument funkcji, która jest po stronie prawej. Wyrażenie `x %>% f(y)` jest równoważne zapisowi `f(x,y)`. Możemy powiedzieć: `x` ma być pierwszym argumentem funkcji `f(y)`, czyli `f(x,y)`. Zastosujmy to podejście do wcześniejszego przykładu.

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

Strefa Eksperta 5.1

Natywny operator potoku `|>`

R od wersji 4.1 ma swój własny operator potoku `|>`. Chociaż idea jest taka sama, to nie zastępuje on jednak operatora `%>%` z pakietu `magrittr`. Przewaga tego drugiego polega na tym, że:

- Funkcja po operatorze nie musi mieć nawiasów, np. `x %>% head` działa, ale `x |> head` już nie.
- Obiekt po lewej stronie obu operatorów jest pierwszym argumentem funkcji po stronie prawej. W wypadku `%>%` jest to domyślna lokalizacja i możemy ją zmienić. Wystarczy kropką wskazać miejsce, np. `x %>% f(z, ., k)` jest równoznaczne z `f(z, x, k)`. Natywny operator nie daje takich możliwości.

Operator natywny może być nieznacznie szybszy, bo jego użycie jest równoznaczne z wywołaniem funkcji. Porównaj:

```
> deparse(substitute( 1:5 |> mean() ))
[1] "mean(1:5)"

> deparse(substitute( 1:5 %>% mean() ))
[1] "1:5 %>% mean()"
```

5.3. Dodawanie zmiennych

Do ramki danych możesz dodawać zmienne lub nadpisywać już istniejące wykorzystując funkcję `mutate()`. Nowe kolumny pojawią się na końcu ramki danych. Ciekawą jej cechą, i wcale nie taką oczywistą, jest możliwość tworzenia zmiennych na bazie aktualnie utworzonych. Pokażę to na przykładzie i poniższej ramce danych.

```
> # Tworzymy ramkę danych: badanie
> pracownik <- c("kierownik", "wykonawczy", "wykonawczy", "kierownik")
> rokUrodz <- c(1987, 1975, 1997, 1970)
> 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     1987  57000      27000
2 wykonawczy     1975  40200      18750
3 wykonawczy     1997  21450      12000
4  kierownik     1970  21900      13200
```

Do ramki danych `badanie` dodamy kolejne zmienne, a powstałą w ten sposób ramkę, zapiszemy pod nazwą `badanieNew`.

- Utworzymy zmienną `wiek` wykorzystując rok urodzenia (`rokUrodz`).
- Utworzymy zmienną `wynDiffUSD`, której wartości są różnicą między obecnym wynagrodzeniem (`wynUSD`) a wynagrodzeniem początkowym (`wynPoczUSD`)
- Zmienną utworzoną w poprzednim punkcie wyrazimy w PLN — przyjmując kurs wymiany na poziomie 3.6 PLN/USD — i zapiszemy jako zmienną `wynDiffPLN`.

Przeanalizuj poniższe rozwiązanie. Zwróć uwagę na wspomnianą przeze mnie cechę, która pozwala utworzyć zmienną `wynDiffPLN` na bazie aktualnie utworzonej `wynDiffUSD`.

```
> badanieNew <- mutate(badanie, wiek = 2019 - rokUrodz,
+                              wynDiffUSD = wynUSD - wynPoczUSD,
+                              wynDiffPLN = 3.6 * wynDiffUSD)
> badanieNew
   pracownik rokUrodz wynUSD wynPoczUSD wiek wynDiffUSD wynDiffPLN
1  kierownik     1987  57000      27000   32      30000     108000
2 wykonawczy     1975  40200      18750   44      21450      77220
3 wykonawczy     1997  21450      12000   22       9450      34020
4  kierownik     1970  21900      13200   49       8700      31320
```

> **WARTO WIEDZIEĆ**
>
> Zdobyta dotychczas wiedza pozwala ci na wykonanie powyższych przekształceń bez użycia funkcji `mutate()` — proponuję wykonać to jako ćwiczenie. Jednak prostota i klarowność zapisu sprawiają, że warto jej używać. Dla sprawdzenia podaję rozwiązanie tego krótkiego ćwiczenia.
>
> ```
> > wiek <- 2019 - badanie$rokUrodz
> > wynDiffUSD <- badanie$wynUSD - badanie$wynPoczUSD
> > wynDiffPLN <-  3.6 * wynDiffUSD
> > (badanieNew <- data.frame(badanie, wiek, wynDiffUSD, wynDiffPLN))
>    pracownik rokUrodz wynUSD wynPoczUSD wiek wynDiffUSD wynDiffPLN
> 1  kierownik     1987  57000      27000   32      30000     108000
> 2 wykonawczy     1975  40200      18750   44      21450      77220
> 3 wykonawczy     1997  21450      12000   22       9450      34020
> 4  kierownik     1970  21900      13200   49       8700      31320
> ```

5.4. Obliczanie statystyk opisowych

5.4.1. Statystyki dla zmiennych ilościowych. W tej części omówimy dwie, ostatnie funkcje przedstawione w tabeli 5.2: `summarise()` i `group_by()`. Jeżeli przeanalizujesz składnię pierwszej, to zobaczysz, że jednym z jej argumentów jest funkcja obliczająca statystykę opisową. W tabeli 5.3 znajdziesz listę takich najczęściej wykorzystywanych funkcji. Zwróć uwagę na dwie kwestie. Pierwsza — argumentem wejściowym jest wektor typu liczbowego, więc możesz te funkcje bezpośrednio wykorzystać do operacji na wektorach i wcale nie potrzebujesz `summarise()`. Druga — wewnątrz `summarise()` używaj tylko tych funkcji, które zwracają jedną wartość. Przynajmniej dwie wartości zwracają funkcje: `summary()`, `cor()` i z domyślnymi ustawieniami `quantile()`.

Tabela 5.3. Funkcje statystyk opisowych

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 różnica: `quantile(x, 0.75)-quantile(x, 0.25)`
`skewness(x, na.rm = FALSE, type = 3)`	Współczynnik asymetrii z pakietu `e1071`. Typ 3 (domyślny) używany jest w programie MINITAB i BMDP, natomiast typ 2 pojawia się w programach IBM SPSS i SAS.
`kurtosis(x, na.rm = FALSE, type = 3)`	Współczynnik spłaszczenia z pakietu `e1071`.
`cor(x, y)`	Współczynnik korelacji Pearsona między zmiennymi `x` i `y`. Jeżeli nie podamy `y`, wtedy `x` musi mieć przynajmniej 2 kolumny. Możemy obliczyć inne współczynniki ustawiając argument `method = "kendall"` lub `"spearman"`. Braki danych obsługujemy argumentem `use` z jedną wartością: `all.obs`, `complete.obs`, `na.or.complete`, `pairwise.complete.obs`.
`summary(x)`	Podaje wartości: największą, najmniejszą, średnią, medianę, pierwszy i trzeci kwartyl, liczbę braków danych

Przypomnę, że widoczne w funkcjach wartości argumentów są domyślne. Jeśli mają mieć taką domyślną wartość, to możesz je pominąć. Przykładowo zapis `mean(x)` jest równoważny zapisowi `mean(x, na.rm = FALSE)`. Jeśli w zbiorze pojawi się choćby jedna brakująca wartość, wtedy wywołanie funkcji z takimi argumentami zwraca `NA`. Działanie to prezentuję poniżej.

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

```
> ## Zmieniamy na: na.rm=TRUE
> ## każąc wyrzucić braki z obliczeń
> mean(x, na.rm = TRUE)
[1] 4
```

Zanim przyjdziemy do zasadniczych przykładów, obliczmy kurtozę, współczynnik asymetrii oraz kwantyle rzędu: 0.25, 0.30, 0.45, 0.95 dla zmiennej saldo rachunku (`balance`) z ramki danych `bank` (zob. tab. 5.1).

```
> ##  Zbiór danych: bank
> library(e1071) # kurtoza i asymetria
> skewness(bank$balance, type = 2)
[1] 8.36031

> kurtosis(bank$balance, type = 2)
[1] 140.752

> # 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
```

Wiesz już, w jaki sposób używać funkcji statystyk opisowych, gdy argumentem wejściowym jest wektor. Teraz możemy przejść do funkcji `summarise()`. W następnym przykładzie policzymy wartości następujących statystyk: średniej, mediany, asymetrii i kurtozy dla zmiennej saldo rachunku (`balance`). Są dwie wersje tego przykładu: z operatorem potokowym i bez niego.

```
> ## Z wykorzystaniem operatora %>%
> bank %>%
+   summarise(srednia = mean(balance),
+     mediana = median(balance),
+     wspAsymetrii = skewness(balance),
+     wspSplaszcz = kurtosis(balance))
  srednia mediana wspAsymetrii wspSplaszcz
1    1362     448         8.36       140.7
```

```
> ## Bez operatora %>%
> summarise(bank,
+     srednia = mean(balance),
+     mediana = median(balance),
+     wspAsymetrii = skewness(balance),
+     wspSplaszcz = kurtosis(balance))
  srednia mediana wspAsymetrii wspSplaszcz
1    1362     448         8.36       140.7
```

Zauważ, że jeśli nie używamy operatora %>%, wtedy na pierwszej pozycji `summarise()` musi znaleźć się nazwa ramki danych. W dalszej kolejności pojawiają sie interesujące nas funkcje, poprzedzone wymyślonymi nazwami kolumn.

Sama funkcja `summarise()` 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 tego samego pakietu. Funkcja ta tworzy grupy składające się z poziomów zmiennych kategorialnych. Przykładowo jeżeli jej argumentem będzie nazwa zmiennej odnoszącej się do edukacji (`education`), wtedy R zapamięta, że funkcję `summarise()` należy zastosować do każdego poziomu tej zmiennej, czyli do każdego poziomu wykształcenia. Tutaj również obowiązuje ta sama składnia: jeżeli nie korzystasz z operatora %>%, wtedy pierwszym argumentem `group_by()` jest nazwa ramki danych. Zobaczmy jak wyglądają statystyki dla salda rachunku w zależności od poziomu edukacji:

```
> ## Z wykorzystaniem operatora %>%
> bank %>%
+   group_by(education) %>%
+   summarise(srednia = mean(balance),
+             mediana = median(balance),
+     wspAsymetrii = skewness(balance))
# A tibble: 4 x 4
  education srednia mediana wspAsymetrii
  <chr>       <dbl>   <dbl>        <dbl>
1 primary     1251.     403         8.85
2 secondary   1155.     392         8.53
3 tertiary    1758.     577         7.43
4 unknown     1527.     568         7.73
```

```
> ## Bez operatora %>%
> krok1 <- group_by(bank, education)
> summarise(krok1,
+     srednia = mean(balance),
+     mediana = median(balance),
+     wspAsymetrii = skewness(balance))
# A tibble: 4 x 4
  education srednia mediana wspAsymetrii
  <chr>       <dbl>   <dbl>        <dbl>
1 primary     1251.     403         8.85
2 secondary   1155.     392         8.53
3 tertiary    1758.     577         7.43
4 unknown     1527.     568         7.73
```

5.4.2. Statystyki dla zmiennych kategorialnych. Podstawowa analiza danych kategorialnych sprowadza się do zliczania wystąpień każdej kategorii. Umiesz już taką operację wykonać, posługując się funkcją `table()`. Będziemy jednak kontynuować pracę z pakietem `dplyr`. Pokażę ci, w jaki sposób oprócz częstości umieszczać dodatkowo w tabeli tzw. częstości względne (odsetki, procenty). Nauczysz się również budować tabele kontyngencji, zwane tabelami krzyżowymi lub wielodzielczymi. Te umiejętności przydadzą ci się, gdy przejdziemy do tworzenia wykresów.

Aby zbudować tabelę składającą się wyłącznie z liczebności, musimy najpierw użyć funkcji grupującej `group_by()`, której argumentami będzie nazwa zmiennej lub zmiennych, dla których tworzymy tabelę. W drugim kroku wykorzystujemy funkcję `summarise()`, wewnątrz której pojawi się funkcja specjalna `n()`. Jej rola sprowadza się do zliczenia wystąpień każdej kategorii zmiennej lub kombinacji kategorii, jeżeli mamy więcej zmiennych. Zapamiętaj, że możesz jej użyć tylko wewnątrz funkcji: `summarise()`, `mutate()` i `filter()`.

Zbudujemy tabelę dla zmiennej edukacja (`education`) oraz tabelę krzyżową dla dwóch zmiennych: edukacja (`education`) i zgoda na lokatę terminową (`y`). Obie zmienne znajdują się w ramce danych `bank`.

```
> ## Tabela 1: education
> bank %>%
+     group_by(education) %>%
+     summarise(ile = n())
# A tibble: 4 x 2
  education   ile
  <chr>     <int>
1 primary    6851
2 secondary 23202
3 tertiary  13301
4 unknown    1857
```

```
> ## Tabela 2: education i y
> bank %>%
+     group_by(education, y) %>%
+     summarise(ile = n())
# A tibble: 8 x 3
# Groups:   education [4]
  education y       ile
  <chr>     <chr> <int>
1 primary   no     6260
2 primary   yes     591
3 secondary no    20752
4 secondary yes    2450
5 tertiary  no    11305
6 tertiary  yes    1996
7 unknown   no     1605
8 unknown   yes     252
```

Zauważ, że w wyniku operacji zwrócona została ramka danych. To oznacza, że możesz do niej dodawać kolejne zmienne za pomocą funkcji `mutate()`. Korzystając z tej możliwości, dodamy odsetki. Zapamiętaj, że sumują się do 1 w obrębie kategorii ostatniej zmiennej, jaka pojawia się wewnątrz funkcji `group_by()`. Która ze zmiennych powinna być ostatnia, zależy wyłącznie od analityka. Przeanalizuj poniższe przykłady pod kątem sumowania do 1.

```
> ## Tabela 1: sumowanie do 1 dla y
> bank %>%
+     group_by(education, y) %>%
+     summarise(ile = n()) %>%
+     mutate(odset = ile/sum(ile))
# A tibble: 8 x 4
# Groups:   education [4]
  education y       ile  odset
  <chr>     <chr> <int>  <dbl>
1 primary   no     6260 0.914
2 primary   yes     591 0.0863
3 secondary no    20752 0.894
4 secondary yes    2450 0.106
5 tertiary  no    11305 0.850
6 tertiary  yes    1996 0.150
7 unknown   no     1605 0.864
8 unknown   yes     252 0.136
```

```
> ## Tabela 2: sumowanie do 1 dla education
> bank %>%
+     group_by(y, education) %>%
+     summarise(ile = n()) %>%
+     mutate(odset = ile/sum(ile))
# A tibble: 8 x 4
# Groups:   y [2]
  y     education   ile  odset
  <chr> <chr>     <int>  <dbl>
1 no    primary    6260 0.157
2 no    secondary 20752 0.520
3 no    tertiary  11305 0.283
4 no    unknown    1605 0.0402
5 yes   primary     591 0.112
6 yes   secondary  2450 0.463
7 yes   tertiary   1996 0.377
8 yes   unknown     252 0.0476
```

Spróbuj, jako ćwiczenie, rozszerzyć którąś z tabel o dodatkową kolumnę, która będzie pokazywała procenty, np. 91.4 (bez znaku %). Dodatkowo zaokrąglij te wartości do jednego miejsca po przecinku.

WARTO WIEDZIEĆ

Oprócz funkcji `n()` w pakiecie `dplyr` znajdują się inne funkcje wyliczające częstości. Dwie przedstawiam w poniższej tabeli.

Tabela 5.4. Funkcje z pakietu `dplyr` obliczające częstości

Funkcja	Opis
`count(rd, zm1)`	Tak samo jak `table()` zlicza wystąpienia każdej kategorii zmiennej `zm1` ramki danych `rd`. Możemy po przecinku dodawać inne zmienne.
`add_count(rd, zm1)`	Zachowuje się podobnie do `count()`, a różnica polega na dodaniu kolumny z liczebnościami do ramki danych `rd`.

Możesz użyć `count()` do szybkiej budowy tabeli z liczebnościami. ł

```
> bank %>% count(education, y)
  education   y     n
1   primary  no  6260
2   primary yes   591
3 secondary  no 20752
4 secondary yes  2450
5  tertiary  no 11305
6  tertiary yes  1996
7   unknown  no  1605
8   unknown yes   252
```

Jeżeli zechcesz dodać odsetki, to zsumują się one do 1 dla całej kolumny. A więc nie działa tutaj zasada sumowania do 1 w obrębie ostatniej kategorii.

5.4.3. Przykład wykorzystania funkcji z pakietu `dplyr`. Na przykładzie zbioru danych `bank` pokażę ci, w jaki sposób poznane funkcje z pakietu `dplyr()` pozwalają na przeprowadzenie analizy wymagającej wielu kroków. W rozwiązaniu, które znajdziesz poniżej, zwróć uwagę na operator potokowy — idealnie wpisuje się w sekwencję tych działań. Spróbuj wykonać poniższe kroki analizy samodzielnie, a dopiero później zapoznaj sie z rozwiązaniem. Lista kroków jest następująca.

1. Interesują nas te obserwacje (klienci), dla których wynik ostatniej kampanii marketingowej (`poutcome`) zakończył się sukcesem lub porażką. Pozostałe poziomy wyrzucamy, bo okazuje się, że takie są.
2. Wszystkie obliczenia wykonujemy w podziale na zmienne: edukacja (`education`), wynik ostatniej kampanii marketingowej (`poutcome`) oraz zgodna na lokatę terminową (`y`).
3. Obliczamy liczebność i procenty dla zmiennych z punktu 2, a dla zmiennej saldo na rachunku (`balance`) liczymy średnią i medianę.
4. W interpretacji i porównaniu otrzymanych statystyk chcemy ograniczyć się tylko do tych wartości, którym odpowiada zgoda na lokatę terminową — dlatego pozostałe usuwamy.
5. Dla ułatwienia porównania sortujmy kolumnę z wartościami procentowymi.
6. Zawężamy 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") %>% # wybierz: sukces lub porażka
+   group_by(education, poutcome, y) %>%
+   summarise(sredniaSaldo = mean(balance),
+             medianaSaldo = median(balance),
+             Liczebnosc = n()) %>%
+   mutate(Procent = 100*Liczebnosc/sum(Liczebnosc)) %>%
+   filter(y == "yes") %>%                                  # weź tylko yes
+   select(-y) %>%                                          # skoro y=yes, to wyrzuć zmienną y
+   arrange(desc(Procent)) %>%                              # posortuj malejąco
+   filter(Procent > 50)
# A tibble: 4 x 6
# Groups:   education, poutcome [4]
  education poutcome sredniaSaldo medianaSaldo Liczebnosc Procent
  <chr>     <chr>           <dbl>        <dbl>      <int>   <dbl>
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
```

Zinterpretujmy pierwszy wiersz: wśród osób o nieznanej edukacji (`education`), które na wcześniejszą kampanię marketingową odpowiedziały 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.

5.5. Rekodowanie zmiennych

Zmienną płeć możemy zakodować na kilka sposobów. Pierwszy — przyjąć: `k` i `m`; drugi — użyć wartości liczbowych, np. `0` i `1`, trzeci — wziąć pełne nazwy: `kobieta` i `mężczyzna`. Jeżeli mamy w zbiorze danych któryś ze sposobów kodowania, możemy chcieć go zmienić. Istnieją również sytuacje, w których niektóre z poziomów chcemy połączyć, np. uważamy, że liczba poziomów dla stanowiska zatrudnienia jest zbyt duża. Właśnie rekodowanie, jak sama nazwa wskazuje, polega na zmianie sposobu kodowania zmiennej.

Niemal każdą operację w R możesz przeprowadzić na wiele różnych sposobów — dotyczy to również rekodowania, które wykonasz za pomocą funkcji tworzącej czynniki `factor()`. Często łatwiej i szybciej poradzimy sobie z tym procesem, jeżeli użyjemy funkcji `recode()` lub `recode_factor()` z pakietu `dplyr`. W poniższej składni pierwszym argumentem jest wektor, w którym przechowywane są wartości zmiennej rekodowanej.

```
recode(zmienna, "obecny_1" = "nowy_1", "obecny_2" = "nowy_2",.default = NULL)
recode_factor(zmienna, "obecny_1" = "nowy_1", "obecny_2" = "nowy_2",
    .ordered = TRUE, .default = NULL)
```

Wartości `obecny` chcemy rekodować (zamienić) na `nowy`. Łańcuch znaków ujmujemy w cudzysłowy. Jeżeli wartością obecną są liczby, to zamiast cudzysłowów, musimy użyć znaku `` ` ``, np: `` `2` ``. Pierwsza z funkcji

zachowuje porządek pierwotnej zmiennej, jeśli jest ona porządkowa. Druga natomiast, ma dodatkowy argument `.ordered`, który pozwala na zmianę tego porządku — porządek ten ustala kolejność, w jakiej wpisujemy wartości `nowy`. Ostatni argument zmienimy z `NULL` na własną wartość, gdy wszystkie pozostałe, niewymienione poziomy mają ją przyjąć. Przykładowo możemy napisać: `.default = "inne"`.

Mamy zmienną typu czynnik (*factor*) o następujących poziomach

```
> ## Losujemy 10 liter ze zbioru: a,b,c
> set.seed(777)
> owoc <- factor(sample(letters[1:3], 10, replace = TRUE))
> owoc
 [1] a b b b c b a a b c
Levels: a b c
```

które chcemy zrekodować na poziomy odpowiadające nazwom owoców: arbuz, banan i czereśnia:

```
> ## Rekodujemy, wykorzystując pakiet dplyr
> library(dplyr)
> recode(owoc, a = "arbuz", b = "banan", c = "czereśnia")
 [1] arbuz     banan     banan     banan     czereśnia banan     arbuz     arbuz     banan
[10] czereśnia
Levels: arbuz banan czereśnia
```

Powyższą funkcję możemy też wykorzystać do połączenia kategorii, np.

```
> ## łączymy kategorie:  a i b - owoc południowy, c - owoc krajowy
> recode(owoc, a = "owoc południowy", b = "owoc południowy", c = "owoc krajowy")
 [1] owoc południowy owoc południowy owoc południowy owoc południowy owoc krajowy
 [6] owoc południowy owoc południowy owoc południowy owoc południowy owoc krajowy
Levels: owoc południowy owoc krajowy

> ## II sposób: przypisuję, aby nie wyświetlać wyniku
> temp <- recode(owoc, c = "owoc krajowy", .default = "owoc południowy")
```

Jeżeli rekodowana zmienna ma wartości liczbowe, wtedy należy ująć je w znaki ``. W poniższym przykładzie mamy zmienną

```
> (wykszt <- factor(sample(1:3, 10, replace = TRUE)))
 [1] 2 1 3 3 3 2 2 2 2 3
Levels: 1 2 3
```

która przyjmuje wartości: 1, 2 i 3 oznaczające odpowiednio wykształcenie: podstawowe, średnie i wyższe. Wiemy, że zmienna wykształcenie jest zmienną porządkową, więc ten fakt uwzględnimy w rekodowaniu i wykorzystamy drugą ze wspomnianych funkcji z dodatkowym argumentem.

```
> recode_factor(wykszt, `1` = "podstawowe", `2` = "średnie", `3` = "wyższe", .ordered = TRUE)
 [1] średnie    podstawowe wyższe     wyższe     wyższe     średnie    średnie    średnie
 [9] średnie    wyższe
Levels: podstawowe < średnie < wyższe
```

W ostatnim przykładzie zrekodujemy numery miesiąca na odpowiadające im kwartały. Najpierw wygenerujemy dane.

```
> # Tworzymy wektor miesiac
> set.seed(123)
> (miesiac <- sample(1:12, 30, replace=TRUE))
 [1]  3  3 10  2  6 11  5  4  6  9 10 11  5  3 11  9 12  9  9  3  8 10  7 10  9  3  4  1 11  7
```

Chcemy zastąpić wartości od 1 do 3: `I_kw`, od 4 do 6: `II_kw` itd. Całą procedurę rekodowania musimy poprzedzić podziałem zmiennej `miesiac` na kategorie odpowiadające kwartałom, używając funkcji `cut()`.

```
> (miesiacKat <- cut(miesiac, breaks = c(0,3,6,9,12))) # breaks - punkty podziału
 [1] (0,3]  (0,3]  (9,12] (0,3]  (3,6]  (9,12] (3,6]  (3,6]  (3,6]  (6,9]  (9,12] (9,12] (3,6]
[14] (0,3]  (9,12] (6,9]  (9,12] (6,9]  (6,9]  (0,3]  (6,9]  (9,12] (6,9]  (9,12] (6,9]  (0,3]
[27] (3,6]  (0,3]  (9,12] (6,9]
Levels: (0,3] (3,6] (6,9] (9,12]
```

Właściwa procedura rekodowania wygląda następująco.

```
> recode_factor(miesiacKat, "(0,3]" = "I_kw", "(3,6]" = "II_kw",
+     "(6,9]" = "III_kw", "(9,12]" = "IV_kw", .ordered = TRUE)
 [1] I_kw   I_kw   IV_kw  I_kw   II_kw  IV_kw  II_kw  II_kw  II_kw  III_kw IV_kw  IV_kw  II_kw
[14] I_kw   IV_kw  III_kw IV_kw  III_kw III_kw I_kw   III_kw IV_kw  III_kw IV_kw  III_kw I_kw
[27] II_kw  I_kw   IV_kw  III_kw
Levels: I_kw < II_kw < III_kw < IV_kw
```

> **WARTO WIEDZIEĆ**
>
> Nie ma jednej, najlepszej, uniwersalnej metody podejścia do problemu. Dlatego nie powinniśmy ograniczać się do znajomości jednego sposobu. Dobrym przykładem jest ostatnie zadanie, w którym numery miesiąca zmieniamy na kwartały. Zapewne lepszym podejściem jest użycie funkcji `factor()`. Zobacz.
>
> ```
> > factor(miesiac, levels = 1:12,
> +     labels = rep(c("I_kw", "II_kw", "III_kw", "IV_kw"), each = 3), ordered = TRUE)
> ```

5.6. Restrukturyzacja danych z pakietem `tidyr`

Poniższe ramki danych zawierają te same dane, ale zorganizowane są w inny sposób. W zależności od rodzaju analizy, dane wejściowe powinny mieć jedną z tych form. Dlatego w tym rozdziale pokażę ci, w jaki sposób strukturę ramki po lewej stronie zamienić na ramkę po stronie prawej — lub odwrotnie.

owoc	sty	lut	mar
kiwi	13	17	21
cytryna	11	20	14

`pivot_longer()` →

← `pivot_wider()`

owoc	miesiąc	sztuki
kiwi	Sty	13
cytryna	Sty	11
kiwi	Lut	17
cytryna	Lut	20
kiwi	Mar	21
cytryna	Mar	14

Do takiej restrukturyzacji wykorzystamy dwie funkcje z pakietu `tidyr`: `pivot_longer()` i `pivot_wider()` o następującej składni:

```
pivot_longer(rd, cols, names_to = "nazwaKol", values_to = "nazwaKolWart")
pivot_wider(rd, id_cols = NULL, names_from = nazwaKol, values_from = nazwaKolWart)
```

Funkcja `pivot_longer()` zbiera kolumny (np. miesiące z tabeli po lewej stronie), a następnie je łączy. To **długa postać** ramki danych. Argumentami funkcji są:

- `cols` — nazwy kolumn ramki `rd`, które chcemy zebrać. Może to być wektor np. `c(sty, lut, mar)`, zakres np. `sty:mar` itp. Negację uzyskamy, dając przed: - albo `!`. Tutaj nie dajemy cudzysłowów, bo odwołujemy się do istniejących nazw kolumn;
- `names_to` — wymyślona nazwa kolumny, w której znają się nazwy zebranych kolumn, np. `names_to = "miesiac"`. A tutaj nazwy nie ma w `rd`, więc będzie cudzysłów;
- `values_to` — wymyślona nazwa kolumny dla wartości, które mają zebrane kolumny, np. `values_to = "sztuki"`.

Przeanalizuj poniższy przykład. Następnie zaproponuj 2 inne sposoby zapisu wartości dla argumentu funkcji `cols`.

```
> ramkaLewa
     owoc sty lut mar
1    kiwi  13  17  21
2 cytryna  11  20  14
```

```
> ramkaPrawa <- pivot_longer(ramkaLewa,
+                      cols = c(sty, lut, mar),
+                      names_to = "miesiac",
+                      values_to = "sztuki")
> ramkaPrawa
# A tibble: 6 x 3
  owoc    miesiac sztuki
  <chr>   <chr>    <dbl>
1 kiwi    sty         13
2 kiwi    lut         17
3 kiwi    mar         21
4 cytryna sty         11
5 cytryna lut         20
6 cytryna mar         14
```

Z kolei funkcja `pivot_wider()` działa dokładnie odwrotnie do funkcji `pivot_longer()`. Rozciąga jedną z kolumn na kilka. To **szeroka postać** ramki danych. Tutaj `id_cols` to kolumny, które identyfikują obserwacje — domyślnie brane są wszystkie. Musimy jednak podać nazwę kolumny `names_from`, którą chcemy rozciągnąć na nowe kolumny (np. `miesiąc`). Za `values_from` bierzemy nazwę kolumny, której wartości wypełnią nowo utworzone kolumny (np. `sztuki`). Zastosuj tę funkcję do tabeli po prawej stronie, a miesiące zostaną rozciągnięte na kilka kolumn. W ten sposób otrzymasz tabelę po lewej stronie. Spójrz na poniższy przykład.

```
> ramkaPrawa
# A tibble: 6 x 3
  owoc    miesiac sztuki
  <chr>   <chr>    <dbl>
1 kiwi    sty         13
2 kiwi    lut         17
3 kiwi    mar         21
4 cytryna sty         11
5 cytryna lut         20
6 cytryna mar         14
```

```
> pivot_wider(ramkaPrawa,
+             names_from = miesiac,
+             values_from = sztuki)
# A tibble: 2 x 4
  owoc      sty   lut   mar
  <chr>   <dbl> <dbl> <dbl>
1 kiwi       13    17    21
2 cytryna    11    20    14
```

Poniższa tabela pokazuje m.in. ceny pewnych produktów w dwóch miastach. Użyj jej, aby utworzyć ramkę danych. Następnie przekształć tę ramkę tak, aby miała wąską postać. Wykonaj również procedurę odwrotną.

produkt	tłuszcz	konserwant	Wrocław	Warszawa
jogurt	3%	tak	3	4
ser	25%	nie	21	28
mleko	3%	nie	2	3

WARTO WIEDZIEĆ

W pakiecie `tidyr` jest użyteczna funkcja `separate()`, która rozdziela kolumnę na kilka kolumn, przyjmując za miejsce podziału wskazany separator. Składnia funkcji jest następująca

```
separate(rd, nazwaKol, nazwyKolNowe, sep = ..., convert = FALSE, remove = TRUE)
```

Dla ramki danych `rd` i wskazanej kolumny `nazwaKol`, tworzone są kolumny o nazwach podanych w postaci wektora `nazwyKolNowe`. Separatorem `sep` może być dowolny znak, np. przecinek (`sep = ","`). Możemy też wpisać liczbę. Wtedy jej wartość odpowiada liczbie znaków tworzących pierwszą kolumnę. Argument `convert` możemy zmienić i ustawić na `TRUE` — nastąpi konwersja typu tworzonych kolumn na bardziej odpowiedni. Bez tej zmiany nowe kolumny będą dziedziczyły typ kolumny dzielonej. Ostatni argument ustawiony na `TRUE` usuwa dzieloną kolumnę. Przeanalizuj poniższy przykład, a później zmień niektóre

argumenty i zaobserwuj różnice.

```
> dat <- data.frame(myData = paste(2018, 1:6, 19:24, sep = "-"))
> ## Ramka danych
> dat
     myData
1 2018-1-19
2 2018-2-20
3 2018-3-21
4 2018-4-22
5 2018-5-23
6 2018-6-24
```

```
> ## Rozdzielamy kolumnę myData
> dat %>% separate(myData, c("rok", "miesiac", "dzien"),
+                  sep = "-", convert = TRUE)
   rok miesiac dzien
1 2018       1    19
2 2018       2    20
3 2018       3    21
4 2018       4    22
5 2018       5    23
6 2018       6    24
```

Odwrotne działanie ma funkcja `unite()`. Jeżeli zastosujesz tę funkcję do utworzonej ramki po prawej stronie: `unite(ramkaPrawaStrona, "myData", sep = "-")`, wtedy otrzymasz ramkę wyjściową `dat`.

5.7.★ Przetwarzanie grupy zmiennych

Funkcje `mutate()` i `summarise()` wymagają od nas wpisania każdej nazwy zmiennej, którą chcemy przetworzyć. Jeśli masz 20 zmiennych, dla których chcesz policzyć średnią, to wewnątrz funkcji `summarise()` musisz aż 20 razy wpisać `mean()` z odpowiednią nazwą zmiennej. Podobna trudność wystąpi, jeśli zamierzasz zestandaryzować każdą z 20 zmiennych wykorzystując `mutate()`. W takich sytuacja przydaj się funkcja `across()`, którą umieszczamy wewnątrz wspomnianych funkcji. Składnia z domyślnymi argumentami ma postać:

```
across(.cols = everything(), .fns, .names = NULL)
```

1. `.cols` — wybór kolumn odbywa się w identyczny sposób, jak to robimy w funkcji `select()`. Zobacz ramkę: warto wiedzieć na str. 52. Różnica dotyczy jedynie sytuacji, gdy podajemy kilka zmiennych. Tutaj musimy połączyć je w wektor, np.: `c(plec, wiek, dzien)`. Zauważ, że domyślnie wybierane są wszystkie kolumny.
2. `.fns` — funkcja albo lista funkcji, które wykonają operację na zmiennych wskazanych w `.cols`. Mamy tutaj kilka możliwości:
   - Wykorzystujemy wbudowane funkcje, np. `mean`, `sd`, `quantile` itp.
   - Budujemy funkcje anonimowe (zob. str. 45).
   - Tworzymy funkcję wcześniej i jej nazwę podajemy jako argument `.fns`.
   - Jeżeli chcemy wykorzystać przynajmniej 2 funkcje, wtedy musimy utworzyć listę. Przykładowo dla dwóch funkcji: `list(mean, sd)`.
3. `.names` — nazwy kolumn po przetworzeniu są takie same jak przetwarzane kolumny. Jeżeli przetwarzamy z wykorzystaniem jednej funkcji, to jest równoważne: `.names = "{.col}"`, gdy jest ich więcej to: `.names = "{.col}_{.fn}"`. Możemy to zmienić, np.: `.names = "moje_{.fn}___{.col}"`.

Zilustrujemy zachowanie funkcji przykładami, których podstawą będzie zbiór danych `airquality` dostępny w R. Wyświetlmy pierwsze dwa wiersze, a następnie zapytajmy, jaki typ ma każda zmienna w kolumnie.

```
> bank <- read_csv2("dane/bankFull.csv") %>%
+         select(age, job, marital, education, duration)
> bank[1:2, ]
# A tibble: 2 x 5
    age job        marital education duration
  <dbl> <chr>      <chr>   <chr>        <dbl>
1    58 management married tertiary       261
2    44 technician single  secondary      151
```

Chociaż typy zmiennych wyświetlane są pod ich nazwami[1], to jednak o nie zapytamy:

[1] Tak się dzieje zawsze, gdy ramka danych jest też obiektem klasy `tbl_df`. Wpisz `class(bank)`.

```
> sapply(bank, typeof)
        age         job     marital   education    duration
   "double" "character" "character" "character"    "double"
```

Jak widzimy, zmienne są typu numerycznego (zmiennoprzecinkowego o podwójnej dokładności) jaki i znakowego. Obliczmy wartości średnie dla zmiennych numerycznych. Wybierzemy więc kolumny, dla których `is.double()` jest `TRUE`.

```
> summarise(bank, across(.cols = where(is.double), .fns = mean)) # możliwe: .cols = c(age, duration)
# A tibble: 1 x 2
    age duration
  <dbl>    <dbl>
1  40.9     258.
```

Jeżeli nie byłoby wbudowanej funkcji liczącej średnią, wtedy możemy utworzyć funkcję anonimową:

```
> summarise(bank, across(where(is.double), ~sum(.x)/length(.x)))
# A tibble: 1 x 2
    age duration
  <dbl>    <dbl>
1  40.9     258.
```

Dla zmiennych typu znakowego podaj liczbę unikalnych elementów. W I sposobie rozwiązania zdefiniuj funkcję anonimowa. W drugim natomiast wykorzystaj wcześniej utworzoną funkcję, która dla wektora typu znakowego zwraca liczbę unikalnych wartości.

Poniżej przedstawiam kolejne przykłady. W pierwszym pokażę, że wewnątrz funkcji `across()` możemy użyć funkcję, która zwraca więcej niż jedną wartość. Zauważ, że `mean()`, `sd()` zwracają jedną, a np. `quantile()` zwraca ich więcej. Właśnie tą ostatnią wykorzystamy, by policzyć kwartyle.

```
> summarise(bank, across(.cols = where(is.double), ~quantile(.x, probs = c(0.25, 0.5, 0.75)),
+                        .names = "kwantyl_{.col}"),
+           rzad_kwantyla = c(0.25, 0.5, 0.75))
# A tibble: 3 x 3
  kwantyl_age kwantyl_duration rzad_kwantyla
        <dbl>            <dbl>         <dbl>
1          33              103          0.25
2          39              180          0.5
3          48              319          0.75
```

W drugim przykładzie zestandaryzujemy wybrane zmienne, zgodnie z formułą: $(x_i - \bar{x})/s$.

```
> mutate(bank, across(.cols = c(age, duration), ~(.x - mean(.x))/sd(.x))) %>% head(4)
# A tibble: 4 x 5
     age job          marital education duration
   <dbl> <chr>        <chr>   <chr>        <dbl>
1  1.61  management   married tertiary    0.0110
2  0.289 technician   single  secondary  -0.416
3 -0.747 entrepreneur married secondary  -0.707
4  0.571 blue-collar  married unknown    -0.645
```

W przykładzie o standaryzacji zmiennych nie uwzględniliśmy możliwości wystąpienia braków danych. Gdyby choć jedna się pojawiła, wtedy zestandaryzowana zmienna wynikowa miałaby same wartości `NA`. Aby się o tym przekonać uruchom:

```
> bankNA <- bank # by nie modyfikować oryginalnego zbioru
> bankNA$age[c(1, 3)] <- NA
```

a następnie linię kodu ze standaryzacją (podmień w niej bank na bankNA). Twoje zadanie polega na tym, by zmodyfikować funkcję standaryzującą i usunąć ten problem. Zrób to w samej funkcji anonimowej. Zaproponuj też drugi sposób: najpierw napisz odpowiednią, by jej nazwę przekazać jako argument funkcji across().

W ostatnim przykładzie przetworzymy odpowiednie kolumny z wykorzystaniem dwóch funkcji. Jak pamiętasz, ich nazwy muszą tworzyć listę. Pojawią się dwa warianty.

```
> summarise(bank, across(.cols = where(is.double), list(srednia = mean, odchStd = sd)))
# A tibble: 1 x 4
  age_srednia age_odchStd duration_srednia duration_odchStd
        <dbl>       <dbl>            <dbl>            <dbl>
1        40.9        10.6             258.             258.
```

W drugim wariancie najpierw definiujemy listę.

```
> Mean_SD <- list(
+     srednia = ~mean(.x, na.rm = TRUE),
+     odchStd = ~sd(.x, na.rm = TRUE)
+ )
> summarise(bank, across(.cols = where(is.double), Mean_SD))
# A tibble: 1 x 4
  age_srednia age_odchStd duration_srednia duration_odchStd
        <dbl>       <dbl>            <dbl>            <dbl>
1        40.9        10.6             258.             258.
```

5.8.★ Łączenie zbiorów za pomocą: left_join() i inner_join()

Operację łączenia zbiorów omówię na przykładzie: złączenia lewego i funkcji left_join() oraz złączenia wewnętrznego i funkcji inner_join(). Obie funkcje pochodzą z pakietu dplyr. Zbudujmy dwie ramki danych: oceny i protokol .

```
album = c("_02", "_03", "_06", "_08", "_4", "_09", "_7")
album2 = c("_02", "_03", "_05", "_08", "_4", "_09", "_7")
imie = c("Marzena", "Paulina", "Ewelina", "Natalia", "Monika", "Marzena","Aleksandra")
ocena = c(5, 2, 3, 4, 5, 4, 3)
protokol <- data.frame(album=album2, imie)[-c(1,4),]
oceny <- data.frame(album, ocena)
```

W pierwszej ramce mamy numer albumu i imię. W drugiej oprócz albumu pojawia się ocena.

```
> protokol
  album       imie
2   _03    Paulina
3   _05    Ewelina
5    _4     Monika
6   _09    Marzena
7    _7 Aleksandra
```

```
> oceny
  album ocena
1   _02     5
2   _03     2
3   _06     3
4   _08     4
5    _4     5
6   _09     4
7    _7     3
```

Naszym pierwszym celem jest dołączenie do ramki protokol kolumny z ocenami, które są w ramce oceny. Dlatego wykorzystamy złączenie lewe.

```
> left_join(protokol, oceny, by = "album")
  album       imie ocena
1   _03    Paulina     2
2   _05    Ewelina    NA
3    _4     Monika     5
4   _09    Marzena     4
5    _7 Aleksandra     3
```

Pierwsze dwa argumenty funkcji to nazwy ramek danych — kolejność jest istotna, bo jeśli zamienimy miejscami protokol z ocena, wtedy do ramki ocena będziemy dołączać protokol. Aby takie złączenie

było w ogóle możliwe, potrzebujemy jakiegoś wspólnego identyfikatora czy klucza. W naszym przykładzie taką rolę pełni kolumna `album`, dlatego pojawia się jako argument `by`. Zauważ, że dla albumu o identyfikatorze `_05` nie mamy oceny, co w konsekwencji prowadzi do `NA`. Widoczne ostrzeżenie jest konsekwencją działania funkcji `data.frame()`, która domyślnie każdy wektor typu znakowego zamienia na wektor typu czynnik. Ponieważ poziomy tego czynnika różnią się między ramkami, dlatego pojawia się ostrzeżenie. Funkcja złączająca, przed wykonaniem operacji, zamienia typ czynnikowy na typ znakowy.

Drugim naszym celem jest utworzenie ramki danych, która będzie zawierała tylko te wiersze, które znajdują się jednocześnie w obu ramkach (taka część wspólna). Wykorzystamy więc funkcję do złączenia wewnętrznego.

```
> inner_join(protokol, oceny, by = "album")
  album       imie ocena
1   _03    Paulina     2
2    _4     Monika     5
3   _09    Marzena     4
4    _7 Aleksandra     3
```

Tym razem wiersz odpowiadający albumowi `_05` nie pojawił się, bo nie ma oceny.

> **WARTO WIEDZIEĆ**
>
> Argument by pozwala również na:
>
> - `by = c("klucz1"= "klucz2")` — jeśli identyfikator/klucz w każdej tabeli ma inną nazwę. W pierwszej tabeli nazwą jest `klucz1`, w drugiej natomiast to `klucz2`.
> - `by = c("zm1", "zm2", "zm3")` — jeżeli kluczem jest więcej niż jedna zmienna. Tutaj mamy 3 zmienne.

5.9. Zadania

Zad. 1. W zbiorze danych `WholesaleCustomers.txt` mamy kolumny odpowiadające: kanałowi dystrybucji, regionom oraz rocznym wydatkom na produkty świeże, mleczne, spożywcze itd.

a) Czy ten zbiór danych można nazwać *tidy*? Jeśli nie, to go przekształć.
b) Oblicz medianę rocznych wydatków dla wszystkich produktów, ze względu na kanał (`Channel`) i region (`Region`). Wariant trudniejszy: nie wykonuj uprzedniej restrukturyzacji ramki danych (zob. rozdz. 5.7).
c) Oblicz wartości tych statystyk opisowych, które pozwalają powiedzieć coś o rozkładzie wydatków na produkty mleczne i spożywcze. Wykonaj obliczenia dla każdego regionu, a wyniki porównaj.
d) Oblicz co dwudziesty percentyl wydatków na wszystkie produkty z uwzględnieniem kanału dystrybucji. Dane posortuj tak, aby powstała tabela pozwoliła porównać regiony ze sobą. Wyciągnij wnioski.
e) Zbuduj tabelę przedstawiającą kanały dystrybucji i region.
f) Czy występuje zależność między produktami ze względu na wielkość wydatków? Posłuż się odpowiednią miarą korelacji.

Zad. 2. Do realizacji poniższych punktów wykorzystaj dane `mieszkania.txt` (udostępnione przez *Fundację Naukową SmarterPoland.pl*).

a) Dodaj do ramki danych kolumnę o nazwie `cena_m2` — to cena za metr kwadratowy.
b) Ile ofert jest w każdym mieście? Policz odsetki.
c) Zbuduj tabelę, w której znajdzie się rozkład liczby ofert ze względu na rok i miasto. Dodaj kolumnę odsetek ofert. W obrębie której kategorii odsetki powinny sumować się do 1?
d) Stwórz taką samą tabelę jak w pkt. c) z tą różnicą, że każde miasto ma ten sam zakres lat. Następnie przekształć powstałą ramkę tak, aby w kolumnach znalazły się nazwy miast, a wartościami były odsetki zaokrąglone do 2 miejsc po przecinku. Wnioski? Spójrz na rezultat:

```
# A tibble: 3 x 4
    rok Krakow Warszawa Wroclaw
  <int>  <dbl>    <dbl>   <dbl>
```

```
1  2009   0.03     0.06    0.02
2  2010   0.16     0.21    0.17
3  2011   0.81     0.73    0.81
```

e) Które miasto ma najdroższe mieszkania? Czy dynamika cen w każdym mieście jest podobna? Jak silna jest zależność między miastami w odniesieniu do zmiany cen mieszkań w czasie?

Zad. 3. 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. Wczytaj plik z danymi `AutoSprzedam.dat` zapisując go w obiekcie o nazwie `auto`.

a) Zmień nazwę pierwszej kolumny `NrOferty`.

b) Stwórz ramkę danych `df_niePolska`, w której oferty nie będą uwzględniały Polski jako kraju pochodzenia samochodu.

c) Stwórz ramkę danych o nazwie `df_kraje3`, w której oferty będą uwzględniały tylko 3 najczęściej występujące kraje pochodzenia (pamiętaj o usunięciu nieużywanych poziomów czynnika). Powstałą ramkę zapisz do pliku i otwórz, pobieżnie sprawdzając poprawność eksportu.

d) Utwórz ramkę danych `df_kolor`, która zawiera tylko samochody w kolorze `czarny-metallic`. Kolumnę odnoszącą się do koloru samochodu usuń.

e) Dodaj do ramki danych `auto` zmienną, która będzie ceną sprzedaży w EUR, z dokładnością do jednego miejsca po przecinku. Kurs wymiany to 4.31 PLN/EUR.

f) Utwórz 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. Uwzględnij następujący fakt: akcyza dotyczy tylko samochodów sprowadzanych z zagranicy. Dlatego musi być równa 0, jeśli `KrajPochodzenia` to Polska. Wyniki dla pierwszych 6 wierszy:

```
  KrajPochodzenia PojemnoscSkokowa CenaPLN Akcyza CenaAkcyza
1          Niemcy             1900   27900    865      28765
2          Polska             2000   28000      0      28000
3          Polska             1781   25500      0      25500
4          Polska             1991   29900      0      29900
5         Francja             2946   29800   5543      35343
6          Niemcy             1800   21400    663      22063
```

g) Ze zbioru danych `auto` usunąć te obserwacje, dla których `RodzajPaliwa` to: `hybryda` lub `napęd elektryczny`. Następnie zrekodować tą zmienną na dwa poziomy: benzyna i olej napędowy.

Zad. 4.** To jest próbka prawdziwych, trudnych problemów przed jakimi stoi analityk. Dlatego warto się z tym zmierzyć. W zbiorze danych `cenyAut2012.Rdata` znajdują się kolumny (np. `Wyposazenie dodatkowe`, `Informacje dodatkowe`, `Adres`), które można wykorzystać i spróbować odpowiedzieć na pytania: (a) czy są jakieś elementy wyposażenia — jeśli tak, to jakie — których obecność przyczynia się do wyższej ceny samochodu; jak wygląda rozkład tej dodatkowej wartości; o ile przeciętnie ta cena jest wyższa (b) czy serwisowanie samochodu i sprzedaż przez pierwszego właściciela wpływają na wartość odsprzedawanego egzemplarza (c) jak długo mają i ile kilometrów przejeżdżają pierwsi właściciele, zanim zdecydują się na sprzedaż; czy są jakieś modele/marki, które sprzedawane są szybciej, a liczba pokonywanych kilometrów jest znacznie większa/mniejsza (d) czy miejscowość/województwo pozwalają cokolwiek powiedzieć o ofercie.

Zanim jednak przystąpimy do właściwej analizy, kolumny należy poddać takim przekształceniom, aby zbiór danych był *tidy*. Poniżej zamieszczam wskazówki, skupiając się głównie na kolumnie `wyposazenie dodatkowe`. Przyda ci się pakiet `stringi`. Jeżeli jakaś funkcja zaczyna się od `stri_` oznacza to, że pochodzi z tego pakietu.

a) Kolumnę `wyposazenie dodatkowe` należy przekształcić na kolumny, a każda powinna odpowiadać jednemu elementowi wyposażenia. To wymaga wielu kroków. Wykorzystaj poznaną funkcję `separate`. Pamiętaj, że musisz podać nazwę każdej, nowej kolumny. W tym celu ustal liczbę elementów wyposażenia dla każdej oferty. Możesz to zrobić, licząc przecinki w każdym wierszu za pomocą funkcji `stri_count(`

`x, fixed = ","`) z pakietu `stringi`, w której `x` jest wektorem, a `fixed` to zliczany element. Choć liczba elementów wyposażenia jest różna, to jednak liczba nowych kolumn w funkcji `separate()` musi odpowiadać maksymalnej liczbie elementów wyposażenia, np. jeśli maksymalna liczba przecinków to 10, wtedy liczba nowych kolumn równa jest 11. Tworzenie nazwy kolumn możesz zautomatyzować — pamiętasz funkcje `paste()`?

b) Masz nowe kolumny z wyposażeniem, ale nie wiesz, czy nie pojawiły się spacje przed lub po. Dla R napis `"ABS"` różni się od `"ABS "`. Usuń je funkcją `stri_trim()`. Zautomatyzuj ten proces. Pomoże ci w tym funkcja `select` (zob. str. 52) i `mutate_all()` (zob. str. **??**).

c) Masz już unikalne elementy wyposażenia?

d) Aby włączyć do zbioru np. nazwy województw, wykorzystać plik `kodyPocztowe.csv`. Pamiętaj, że kod pocztowy nie jest unikalnym identyfikatorem nazwy miejscowości — ten sam kod może odpowiadać innym miejscowościom. Z dodatkowych funkcji może ci się przydać: `stri_trans_tolower()`, która zamienia duże litery na małe, czy też funkcja `stri_trans_general()` usuwająca znaki diakrytyczne. W tej ostatniej za argument `id` przyjmij: `id = "Latin-ASCII"`. Jeżeli chcesz usunąć pewne elementy napisu, wtedy do lokalizacji służy `stri_locate()`, a do usuwania `stri_replace()` — obie zapewne będą z argumentem `fixed`.

ROZDZIAŁ 6 Wizualizacja danych z pakietem ggplot2

Możliwości R są ogromne w zakresie wizualizacji danych. 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. Przykładowo w ramach systemu `grid` powstał pakiet `lattice`, który pozwala budować wykresy z wieloma zmiennymi (wykresy warunkowe). Zyskał on popularność, gdyż w odróżnieniu od systemu tradycyjnego formatuje elementy wykresu. W rezultacie taki wykres możemy bezpośrednio użyć np. w publikacji. Na bazie systemu `grid`, w 2005 roku, powstał również pakiet `ggplot2`. Jego twórca, Hadely Wickham, wziął dobre elementy systemu tradycyjnego i pakietu `lattice`, a następnie udoskonalił je. Poza tym wykorzystał gramatykę grafiki Wilkinsona — stąd w początkowej nazwie pakietu widzimy gg. Gramatyka to zbiór reguł mówiących o zasadach tworzenia grafiki. Odpowiada ona na pytanie: czym jest grafika statystyczna.

6.1. Schemat budowy wykresu

Wykresy budujemy, nakładając kolejne warstwy. Do nich zaliczamy m.in.: estetykę, geometrię, skale, współrzędne i panele. Takie sekwencyjne podejście do rysowania wykresu powoduje, że kolejna warstwa może zasłonić bądź zmienić elementy warstwy wcześniejszej. Poniżej znajdziesz krótki opis każdej z warstw[1]. Zwróć szczególną uwagę na pierwsze dwie warstwy, które są niezbędne do utworzenia wykresu. Jeśli jednak chcesz mieć większą kontrolę nad ostatecznym wyglądem wykresu, zapoznaj się z pozostałymi — potraktuj je jako materiał nieobowiązkowy.

Mapowanie estetyk

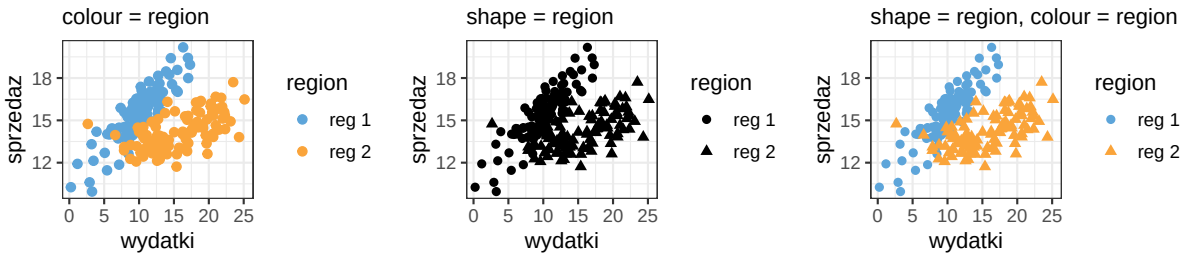
Pierwszym elementem jest tzw. estetyka (*aesthetic*). Odwzorowuje dane w atrybuty wykresu takie jak: kolor, kształt, rozmiar. Mówi nam o roli jaką każda zmienna pełni na wykresie. Przykładowo, jedna zmienna będzie odwzorowywana na osi X, druga na osi Y, a trzecia będzie odwzorowana w postaci koloru (a może kształtu lub rozmiaru) i umieszczona w legendzie.

Aby utworzyć wykres, użyj funkcji `ggplot()`. Pierwszym jej argumentem jest nazwa ramki danych. Kolejne odnoszą się do elementów estetycznych — przydziel role poszczególnym zmiennym, zgodnie z poniższą składnią. W miejsce kropek wstaw nazwy zmiennych ramki danych. Zwróć również uwagę na skrót *aes* pochodzący od angielskiego słowa *aesthetic*.

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

Zapewne nie wszystkie elementy estetyki będziemy wykorzystywać jednocześnie. Rozważmy prosty przykład, w którym chcemy wizualizować relacje między wydatkami na reklamę a sprzedażą. W tej sytuacji przyjmujemy, że na osi X będzie `wydatki`, a więc `x = wydatki`. Z kolei na osi Y znajdzie się zmienna `sprzdaz`. To wystarczy do stworzenia prostego wykresu. Załóżmy jednak, że firma działa w 3 województwach i ten fakt również powinniśmy uwzględnić na wykresie. Ta zmienna znajdzie się zapewne w legendzie, ale pojawia sie pytanie, który element estetyczny powinniśmy wybrać. To jest ściśle powiązane z drugą warstwą (elementami geometrycznymi), o której piszę poniżej. Bo jeśli zdecydujemy się na punkty, jako element geometryczny, wtedy możemy je różnicować kolorem (`colour`) albo kształtem (`shape`). Mamy również możliwość wyboru obu estetyk. Spójrz na poniższe warianty wykresów.

[1] Dokumentacja `ggplot2` jest również dostępna pod oficjalnym adresem: `http://ggplot2.tidyverse.org/reference/`



Elementy geometryczne

Drugim elementem (warstwą) jest atrybut geometryczny. Mówi nam o tym, w jakie elementy geometryczne będą odwzorowywane nasze zmienne. Możemy wybrać punkty, linie, słupki itp. Funkcje odpowiadające geometrii zaczynają się od `geom_`. Drugi człon, który musimy dodać, odpowiada nazwie elementowi geometrycznemu. Punkty które widzisz na powyższych wykresach dodałem wykorzystując funkcję `geom_point()`. I to dosłownie — użyłem znaku +. W taki sposób będziemy dodawać kolejne warstwy. Spójrz na poniższą składnię, której wynikiem jest utworzony wcześniej wykres, widoczny po prawej stronie.

```
ggplot(wydSprz, aes(x = wydatki, y = sprzedaz, shape = region, colour = region)) +
    geom_point()
```

Zwróć uwagę na brak argumentów funkcji `geom_point()`. Zazwyczaj elementy geometryczne ich nie wymagają. Jeśli jednak chcesz zmienić np. rozmiar punktu, wtedy musisz zmienić wartość domyślną odpowiedniego argumentu. Listę najczęściej wykorzystywanych elementów geometrycznych zamieszczam w poniższej tabeli. Ich omówieniem zajmę się w dalszej części opracowania. Argumentami opcjonalnymi nie musisz się przejmować — potraktuje je jako materiał nieobowiązkowy. Jeśli jednak chcesz mieć kontrolę nad ostatecznym wyglądem wykresu, spróbuj zapamiętać następujący schemat. Argument `size` odpowiada wielkości punktu lub grubości linii — przyjmuje wartości rzeczywiste dodatnie (np. 0.75, 2). Kolor linii lub punktu zmienisz używając argumentu `colour`. Podobną funkcję pełni `fill`, ale odpowiada on wypełnieniu czegoś — może to być słupek, pole pod krzywą albo punkt z pustym środkiem (np. okrąg, trójkąt); linii wypełnić nie możemy. Rodzaj linii definiuje argument `linetype`. Choć będę o tym pisał, to wiedz, że możesz podstawić wartości naturalne od 0 do 6. Rodzaj punktu `shape` ustawisz wpisując liczbę naturalną od 0 do 25. Pojawiająca się wszędzie `alpha` odpowiada nasyceniu koloru i przyjmuje wartości z przedziału $[0,1]$, gdzie największa wartość oznacza całkowite pokrycie. Przykład: `geom_point(size=4, shape=20, colour="blue", alpha=0.3)`.

Tabela 6.1. Funkcje elementów geometrycznych

Funkcja	Rysuje	Argumenty opcjonalne
`geom_point()`	Punkty	`size`, `shape`, `colour`, `fill`, `alpha`
`geom_line()`	Linie	`size`, `linetype`, `colour`, `alpha`
`geom_vline(xintercept = ...)`	Linie pionowe	`size`, `linetype`, `colour`, `alpha`
`geom_hline(yintercept = ...)`	Linie poziome	`size`, `linetype`, `colour`, `alpha`
`geom_smooth()`	Krzywą wygładzoną	`method`, `se`, `size`, `linetype`, `colour`, `alpha`
`geom_segment(aes(xend=..., yend=...))`	Linie od ... do ...	`size`, `linetype`, `colour`, `alpha`
`geom_bar()`, `geom_col()`	Słupki	`position`, `linetype`, `colour`, `fill`, `alpha`
`geom_histogram()`	Histogram	`bins`, `linetype`, `colour`, `fill`, `alpha`
`geom_density()`	Gęstość	`size`, `linetype`, `colour`, `fill`, `alpha`
`geom_boxplot()`	Pudełko-wąsy	`size`, `linetype`, `colour`, `fill`, `alpha`, `coef`, `outlier.size`, `outlier.shape`, `outlier.colour`, `outlier.fill`, `outlier.alpha`
`stat_function(fun = ...)`	Funkcję	`args=list(...)`, `size`, `linetype`, `colour`, `alpha`
`stat_ecdf()`	Dystrybuantę	`size`, `linetype`, `colour`, `alpha`
`stat_qq()`	Punkty kwantyl-kwantyl	`size`, `shape`, `colour`, `fill`, `alpha`
`stat_qq_line()`	Linię kwantyl-kwantyl	`size`, `linetype`, `colour`, `alpha`

Obok elementów geometrycznych, w tabeli 6.1, widzisz też elementy statystyczne, które zaczynają się od `stat_`. Ich zadaniem jest przekształcanie zmiennych z wykorzystaniem elementów statystyki i odwzorowanie wyniku w elementy geometryczne.

Skale

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, zakresie wartości danych itd. Skoro argumenty funkcji z tabeli 6.1, zapytasz, pozawalają zmieniać kolorystykę, to po co mi kontrola nad skalami? Musisz jednak wiedzieć, że te argumenty ustawiają jeden i ten sam kolor dla danego obiektu geometrycznego. Przypomnij sobie wykres o wydatkach i sprzedaży, na którym mieliśmy dwa kolory punktów zależne od regionu. Zmieniłem je z domyślnych, wykorzystując skale — inaczej się nie da.

Wiedz, że skale są elementem opcjonalnym. Zmień je wtedy, gdy domyślne ustawienia ci nie odpowiadają. Lista funkcji jest długa, więc zaprezentuję tylko niektóre. Dodatkowo ograniczę się do pewnego schematu. Ponieważ funkcje składają się z 3 członów oddzielonych znakiem podkreślenia — a pierwszy już znasz (`scale_`) — więc w poniższej tabeli pojawią się nazwy dwóch następnych. W ramach każdego wiersza utworzysz tyle funkcji, ile jest możliwych powiązań między drugim a trzecim członem. Ponadto zapamiętaj, że wybór elementu dla drugiego członu związany jest z tym, co chcesz zmienić. Jeśli będzie to rozmiar linii bądź punktu wybierzesz `size`, a jeśli wypełnienie słupka to `fill` itd. Rozważmy jeszcze przykład tworzenia funkcji. W ostatnim wierszu mamy 2 możliwości w drugim członie i tyle samo w trzecim, więc utworzymy 4 funkcje: (1) `scale_x_continuous` (2) `scale_x_discrete` (3) `scale_y_continuous` (4) `scale_y_discrete`.

Tabela 6.2. Funkcje dla skal

Drugi człon	Trzeci człon	Szczegóły
`colour, fill`	`brewer`	Kolorystyka z pakietu `RColorBrewer`. Wymagany argument `palette` i znajomość jego wartości, np. `scale_colour_brewer(palette = "Set1")`
`colour, fill`	`manual`	Ręcznie ustawiany argument koloru `values`. Wymagana znajomość kodu szesnastkowy koloru (HEX), np. zmiana dwóch kolorów: `scale_fill_manual(values = c("#69b4ff", "#377eb8"))`
`size, shape, linetype`	`manual`	Ręcznie ustawiany argument `values` dla rozmiaru, kształtu lub typu linii, np. `scale_linetype_manual(values = 1)`
`x, y`	`continuous, discrete`	Ręcznie ustawiany przynajmniej jeden z argumentów: `breaks`, `labels`, `limits`, np. `scale_y_continuous(breaks = seq(1, 13, 3), limits = c(4, 10))`. Dane poza zakresem są ustawiana na `NA`.

Do skal włączymy jeszcze dwie ważne funkcje.

```
labs(x = ..., y = ..., fill = ..., colour = ..., shape = ...)
ggtitle(label = ..., "podtytul")
```

Pierwsza zmienia domyśle opisy dla osi OX i OY oraz legendy. W wypadku opisu legendy wybieramy te elementy estetyczne, które użyliśmy w funkcji `ggplot()`. Druga z funkcji dodaje tytuł wykresu (`label`) oraz opcjonalnie podtytuł. Przykładowo dla wykresu o wydatkach i sprzedaży, tego po lewej stronie, wpisałem:

```
labs(x = "wydatki", y = "sprzedaz", colour = "region") +
ggtitle(label = NULL, "region") # usuń duży tytuł (dlatego NULL), dodaj podtytuł
```

Współrzędne

Współrzędne stanowią kolejny element. Zaczynają się od `coord_`. Szczególnie przydadzą nam się trzy funkcje, które pozwalają definiować zakresy dla osi X i Y, ustawiać proporcję między osiami oraz

zamieniać osie miejscami (transponować). Widoczne wartości argumentów w poniższej tabeli są domyślne.

Tabela 6.3. Funkcje dla współrzędnych

Funkcja	Opis
`coord_cartesian(xlim = NULL, ylim = NULL, expand = TRUE)`	Zmienia zakres współrzędnych X i Y. Dane poza zakresem są niewidoczne, ale nie jak w wypadku skal ustawione na `NA`. Argument `expand = TRUE` dodaje niewielkie wartości do zakresu.
`coord_fixed(ratio = 1, xlim = NULL, ylim = NULL, expand = TRUE)`	Zmienia proporcje `ratio` między osiami. Dodatkowo pozwala na zmianę zakresu współrzędnych.
`coord_flip(xlim = NULL, ylim = NULL, expand = TRUE)`	Zamienia współrzędne miejscami (transponuje). Dodatkowo pozwala na zmianę zakresu współrzędnych.

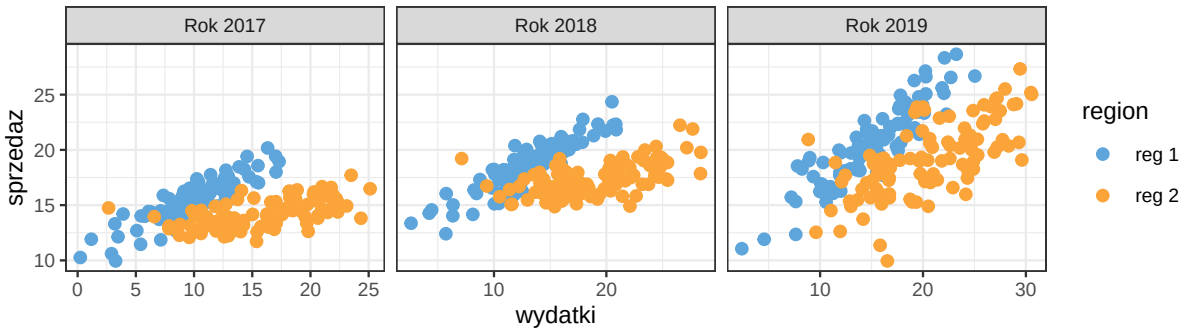
Panele

Ostatni element (*facets*) pozwala nam przedstawić wiele wykresów cząstkowych w tzw. panelach. Idea polega na tym, że dane dzielone są ze względu na poziomy zmiennej kategorialnej, i dla każdego poziomu tworzony jest osobny wykres. Możemy z tego skorzystać, zamiast zmienną przenosić do tzw. legendy. Częściej jednak posługujemy się tym rozwiązaniem, gdy w legendzie są już zmienne, a chcemy dodać kolejną. Wykorzystamy pierwszą z poniższych funkcji.

```
facet_wrap(~ zm, nrow = NULL, ncol = NULL, scales = "fixed")
facet_grid(zm_1 ~ zm_2, scales = "fixed", space = "fixed")
```

Wymaganym argumentem jest nazwa zmiennej kategorialnej lub dyskretnej `zm`. Kolejne opcjonalne argumenty pozwalają kontrolować liczbę wykresów w wierszu (`nrow`) albo kolumnie (`ncol`). Domyślnie skale osi są takie same w wszystkich panelach (`fixed`). Możemy to zmienić, przyjmując: `free_x`, `free_y` lub `free`. Wtedy skale będą różne dla wybranej osi albo dla wszystkich osi. Poniższy wykres otrzymałem dodając warstwę, która widzisz poniżej. Choć różne skale dla wydatków raczej nie mają tutaj uzasadnienia, to zrobiłem to celowo — zwróć uwagę na osie X i Y w 3 panelach.

```
facet_wrap(~ rok, scales = "free_x")
```



Z kolei funkcja `facet_grid()` przedstawia wykresy w postaci macierzowej. W wierszu znajdą się poziomy zmiennej `zm_1`, w kolumnie natomiast poziomy zmiennej `zm_2`.

6.2. Wybrane wykresy

W tej części praktycznej pokażę ci, w jaki sposób budować najpopularniejsze wykresy. Do takich zaliczam wykresy: rozrzutu (punktowe), liniowe, słupkowe. Do tej grupy włączę też wykresy pozwalające przedstawić rozkład zmiennej ilościowej za pomocą: funkcji gęstości, dystrybuanty, rozkładu kwantyl-kwantyl czy wykresu pudełko-wąsy. Wszystkie funkcje, których będziemy potrzebować, zamieściłem w tabeli 6.1 na stronie 70. Staram się nie używać argumentów opcjonalnych i ograniczam się do dwóch pierwszych warstw (estetyki i geometrii). Zainteresowanych kolejnymi warstwami i możliwościami formatowania wykresów odsyłam do materiałów dodatkowych, które zamieściłem w rozdziale **??**. Nieraz zmuszony jestem użyć argumentów opcjonalnych czy też warstw. Dobrym przykładem argumentu

opcjonalnego, który zmieniam, jest rozmiar punktu (`size`). Zdarza się, że zmienna kategorialna ma wiele kategorii, których etykiety nie mieszczą się na osi X. Wtedy zamieniam współrzędne miejscami dodając warstwę `coord_flip()`. Bez tych zabiegów wykresy mogą być nieczytelne. Kolejnym, obowiązkowym elementem są podpisy osi. Z tego względu te elementy formatowania, które pojawią się w tym rozdziale (6.2), potraktuj jako obowiązkowe.

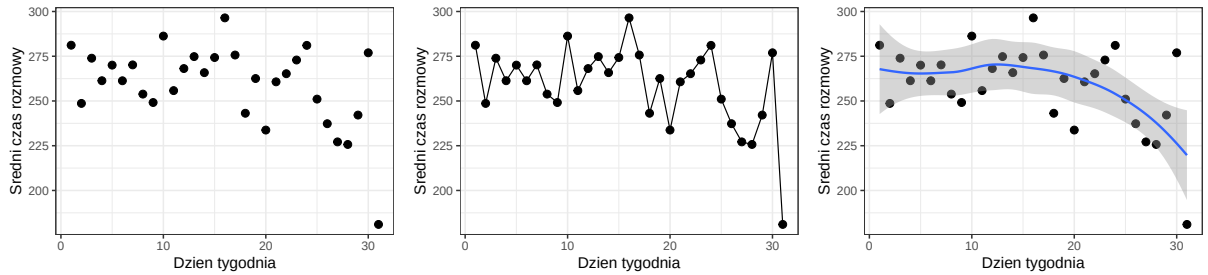
Muszę jeszcze wspomnieć o związku między kodem a wykresami, które widzisz. Zawsze do finalnego wykresu dodaję warstwę odnoszącą się do tematu `theme_bw()` — aby usunąć szare tło — i zmieniam domyślną kolorystykę. Tego nie zobaczysz w kodzie, bo celowo tego nie pokazuję. Dlatego jeśli uruchomisz zamieszczone tutaj skrypty, nie otrzymasz identycznego wykresu. Pamiętaj również o wczytaniu odpowiednich pakietów — sugeruję wczytać zbiorczy pakiet `tidyverse`. W tym rozdziale będziemy pracowali z ramką danych `bank`, którą omówiłem na stronie 51.

6.2.1. Punkty i linie. Już wiesz, że wykresy tworzymy nakładając kolejne warstwy. Tym samym możemy najpierw narysować punkty, później je połączyć linią, a ostatecznie nanieść krzywą wygładzoną. W poniższym przykładzie wykorzystamy odpowiednie funkcje z tab. 6.1. Użyjemy ramki danych `dayCzas`, którą utworzymy na podstawie ramki `bank`.

```
> ## Ramka danych: bank; tworzymy ramkę dayCzas
> dayCzas <- bank %>%
+    group_by(day) %>%
+    summarise(czas = mean(duration))
> head(dayCzas)
# A tibble: 6 x 2
    day  czas
  <int> <dbl>
1     1  281.
2     2  249.
3     3  274.
4     4  261.
5     5  270.
6     6  261.
```

W powyższej ramce danych mamy średni czas rozmowy w każdym dniu miesiąca. Chcemy to przedstawić na wykresie. W pierwszym kroku określamy tzw. estetykę — która zmienna będzie na jakiej osi. Niech na osi X będzie zmienna `day`, a na osi Y znajdzie się zmienna `czas`. Drugi krok to wybór elementu geometrycznego. Stwórzmy 3 różne wykresy.

```
> p1 <- ggplot(dayCzas, aes(x = day, y = czas)) +
+ geom_point(size = 3) +
+ labs(x = "Dzien tygodnia", y = "Sredni czas rozmowy")
> p1                  # wykres po lewej stronie
> p1 + geom_line()    # wykres środkowy
> p1 + geom_smooth()  # wykres po prawej stronie
```



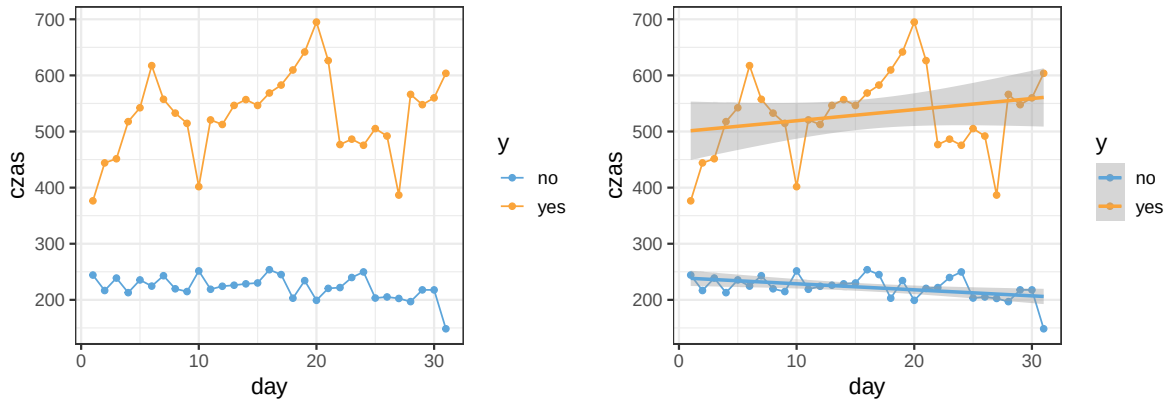
Jak widzisz, pierwszy wykres składa się tylko z punktów. Aby go wyświetlić wpisujemy `p1`. Do tego wykresu dodajemy kolejną warstwę geometryczną — linię. To zapewne ułatwi nam interpretację. Następnie do punktów dodajemy inny element geometryczny — krzywą wygładzoną. W ten sposób otrzymujemy ostatni wykres. Czy możemy do punktów dodać zarówno linię jak i krzywą wygładzoną? Spróbuj dodać do siebie wszystkie 3 elementy geometryczne. Proponuję również zmienić krzywą wygładzoną na prostą pisząc: `geom_smooth(method = "lm")`.

Zbudujemy wykres podobny do poprzednich wykresów. Różnica będzie polegać na dodaniu zmiennej kategorialnej y, odpowiadającej zgodzie na lokatę terminową. W konsekwencji pojawią się dwie linie i dwie grupy punktów odpowiadające poziomom dodanej zmiennej: yes, no. Zanim jednak przystąpimy do budowy wykresu, musimy przygotować dane.

```
> dayCzas2 <- bank %>%
+    group_by(day, y) %>%
+    summarise(czas = mean(duration))
> head(dayCzas2, 4)
# A tibble: 4 x 3
# Groups:   day [2]
    day y      czas
  <int> <chr> <dbl>
1     1 no     244.
2     1 yes    376.
3     2 no     217.
4     2 yes    444.
```

Dodatkowa zmienna wymaga od nas dodania kolejnego elementu estetycznego. Jeśli chcemy dokonać rozróżnienia między kategoriami zmiennej y za pomocą koloru, wtedy napiszemy: colour=y. Nie jest to jedyna możliwość. Możemy np. dobrać też różne kształty dla punktów, co osiągniemy, pisząc shape=y. Niech nie zmyli nas nazwa tej zmiennej. Nie ma ona nic wspólnego z elementem estetycznym y=.... To jest zwykły zbieg okoliczności. Decydujemy się na pierwsze rozwiązanie i wybieramy kolor jako podstawę rozróżnienia kategorii.

```
> p2 <- ggplot(dayCzas2, aes(x = day, y = czas, colour = y)) +
+       geom_point() + geom_line()
> p2                              # wykres po lewej
> p2 + geom_smooth(method = "lm") # wykres po prawej
```



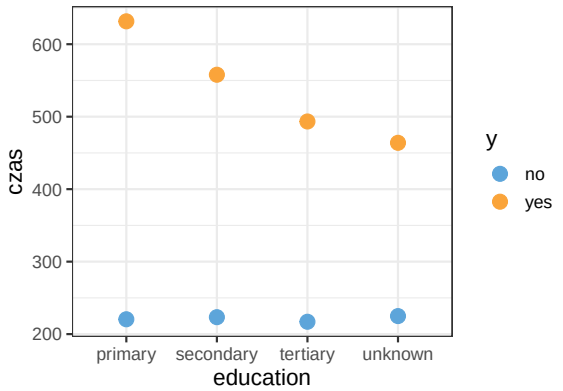
Dodając kolejną warstwę w postaci linii, otrzymaliśmy zamierzony efekt, tzn. punkty zostały połączone linią poziomą. Ale zastanów się: dlaczego R nie połączył tych punktów linią pionową? Przecież technicznie też jest to możliwe. O tym zadecydował typ liczbowy zmiennej day i zmiennej czas. R w tym wypadku nie miał żadnych wątpliwości. Wobec tego co się stanie, jeśli zamiast zmiennej ilościowej na osi *X* będziemy mieli zmienną kategorialną? Przeanalizuj poniższą ramkę danych i odpowiadający jej wykres punktów. Jak myślisz, jaki będzie efekt dodania linii?

Zapamiętaj 6.1

Aby ułatwić analizę wykresów, zrezygnowałem z opisów osi. Pamiętaj, że trzeba to zrobić, np. dodając:

```
labs(x = "Poziom edukacji", y = "Średni czas rozmowy", colour = "Depozyt")
```

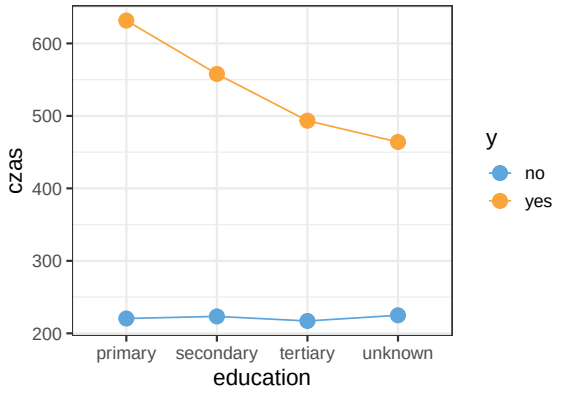
```
> eduCzas <- bank %>%
+    group_by(education, y) %>%
+    summarise(czas = mean(duration))
# A tibble: 8 x 3
# Groups:   education [4]
  education y      czas
  <chr>     <chr> <dbl>
1 primary   no     220.
2 primary   yes    632.
3 secondary no     223.
4 secondary yes    558.
5 tertiary  no     217.
6 tertiary  yes    493.
7 unknown   no     225.
8 unknown   yes    464.
```



W zależności od tego co chcemy zaakcentować na wykresie, mamy do wyboru dwie możliwości. Pierwsza — łączymy punkty linią pionową. W ten sposób zwracamy większą uwagę na różnice między decyzjami odnośnie depozytów. Interpretując taki wykres możemy powiedzieć, że wraz ze wzrostem poziomu edukacji, skraca się dystans między czasem rozmów osób zgadzających się i niezgadzających się na lokatę terminową. Druga możliwość — łączymy punkty linią poziomą, gdy bardziej chcemy zaakcentować zmianę czasu rozmowy w zależności od poziomu wykształcenia. W tej sytuacji powiemy, że czas rozmów osób godzących się na lokatę terminową wyraźnie spada wraz ze wzrostem wykształcenia. Tego nie można powiedzieć o tych, którzy nie zdecydowali się na taką lokatę. Niezależnie od wykształcenia, ten czasu utrzymuje się na podobnym poziomie.

Powyższe uwagi prowadzą nas do oczywistego wniosku: musimy w elemencie estetycznym `group` podać nazwę tej zmiennej, w obrębie której punkty mają być połączone. Decydując się na drugą możliwość tą zmienną jest `y`.

```
> ggplot(eduCzas, aes(x = education,
+                     y = czas,
+                     colour = y,
+                     group = y)) +
+ geom_line() +
+ geom_point(size = 4)
```

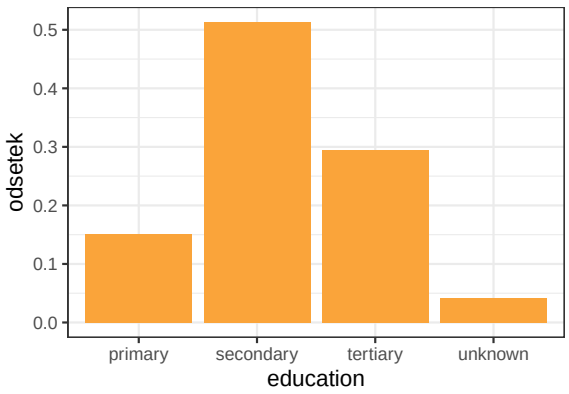


6.2.2. Wykresy słupkowe. Wykres słupkowy rysujemy nanosząc warstwę geometryczną za pomocą `geom_bar()` albo `geom_col()`. Różnice między tymi funkcjami wyjaśniam w ramce: warto wiedzieć. Skupimy się tylko na drugiej funkcji i następującej strategii tworzenia wykresów słupkowych. Jeśli chcemy wizualizować statystyki zmiennych kategorialnych tj. liczebność, odsetki lub procenty, wtedy musimy stworzyć odpowiednią tabelę. Tworzyliśmy je w rozdz. 5.4.2 i 5.4.3. Chodzi o to, aby każdej wartości odpowiadał jeden słupek, bo tego wymaga funkcja `geom_col()`. Dla tak przygotowanej ramki budujemy wykres.

Wykorzystamy tę strategię i zbudujemy wykres, który będzie przedstawiał rozkład edukacji. Przyjmijmy, że interesującą nas statystyką jest odsetek. Poniżej tworzymy tabelę `tabEdu`, a następnie wykres. Zwróć uwagę na elementy estetyczne. Zamiana osi nie spowoduje, że otrzymasz to, o czym myślisz. Jeśli osie chcesz zamienić miejscami, to użyj warstwy odpowiedzialnej za współrzędne: `coord_flip()`.

```
> tabEdu <- bank %>% count(education) %>%
+   mutate(odsetek  = n/sum(n))
> tabEdu # ramka do zwizualizowania
  education     n odsetek
1   primary  6851  0.1515
2 secondary 23202  0.5132
3  tertiary 13301  0.2942
4   unknown  1857  0.0411

> p4 <- ggplot(tabEdu, aes(x = education, y = odsetek)) +
+       geom_col()
```



Warto wiedzieć

Budowa wykresów słupkowych: `geom_bar()` a `geom_col()`

Niezależnie od tego, którą z poniższych funkcji zastosujesz, otrzymasz identyczny rezultat.

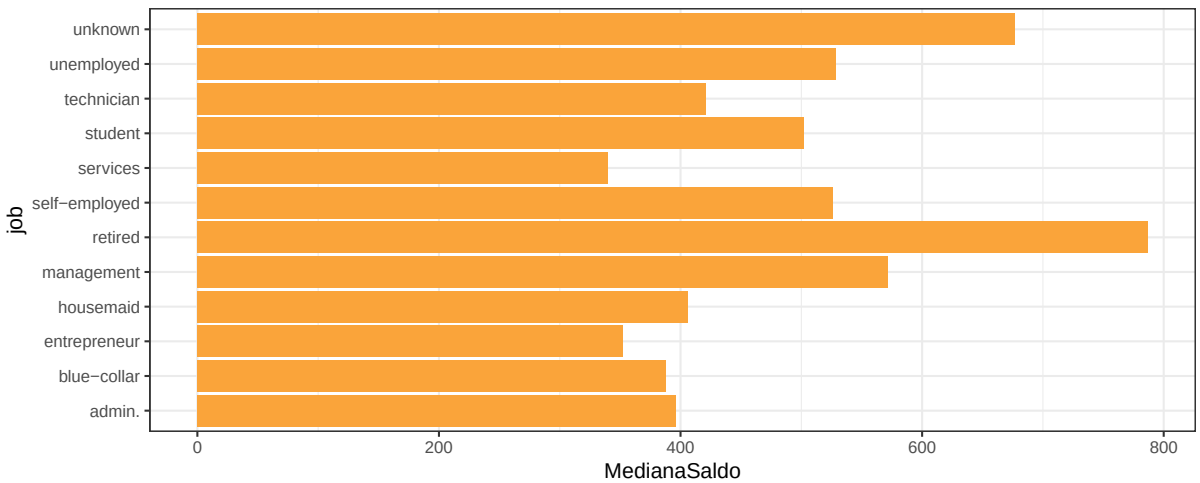
```
geom_col()
geom_bar(stat = "identity")
```

Bez opcjonalnego argumentu druga funkcja najpierw wywołuje funkcję zliczającą wystąpienia wszystkich poziomów zmiennej kategorialnej, a dopiero później tworzy słupki odpowiadające tym liczebnościom. Tak wywołana funkcja wymaga też oryginalnej, nieprzetworzonej ramki bank. Zazwyczaj jesteśmy zainteresowani wartościami względnymi (odsetkami, procentami), więc musimy je policzyć. To z kolei narzuca na nas konieczność wpisania argumentu opcjonalnego. Czy nie szybciej będzie, jak użyjemy `geom_col()`?

Wysokość słupka może odpowiadać wybranej statystyce dla zmiennej ciągłej. Jako miarę agregacji możemy wziąć średnią, medianę, kwantyl itd. Zobaczmy jak kształtuje się mediana salda rachunku (`balance`) w zależności od kategorii zatrudnienia (`job`). Aby kategorie na osi X nie zachodziły na siebie, zamienimy osie miejscami, dodając `coord_flip()`.

```
> balJob <- bank %>%
+    group_by(job) %>%
+    summarise(MedianaSaldo = median(balance))
> head(balJob, 3)
# A tibble: 3 x 2
  job          MedianaSaldo
  <chr>               <dbl>
1 admin.                396
2 blue-collar           388
3 entrepreneur          352

> ggplot(balJob, aes(x=job, y=MedianaSaldo)) + geom_col() + coord_flip()
```



Jeśli zmienna kategorialna ma wiele poziomów, rozważ wykres punkty zamiast wykresu słupkowego. Taki wykres już umiesz zbudować. W poniższym przykładzie, który bazuje na ramce `balJob`, uwzględniam dwie nowe funkcje. Pierwsza `reorder()` sortuje kategorie zmiennej dyskretnej `jakaZm` w kolejności `kolejnosc`. Taki zabieg na pewno ułatwi nam interpretację wykresu. Oczywiście ma to sens wtedy, gdy zmienna zmierzona jest na skali nominalnej. Zauważ, że do poprzedniego wykresu słupkowego również mogliśmy ją użyć. Druga funkcja pojawiła się w tabeli 6.1, ale jej nie omówiłem. Chodzi mi o `geom_segment()`, która pozwala dodać linie pionowe lub poziome w zakresie od do. W naszym przykładzie te linie będą pełniły rolę linii wiodących — ułatwiają powiązanie punktu z kategorią. Musimy tylko podać gdzie się kończą x i y, bo swój początek mają już w narysowanym, czarnym punkcie.

```
> p5 <- ggplot(balJob, aes(x = MedianaSaldo, y = reorder(job, MedianaSaldo)))
> p5 <- p5 + geom_point(size=3)
> p5  + labs(x = "Mediana salda", y = "Kategoria zatrudnienia") # wykres po lewej stronie
> p5 + geom_segment(aes(yend = job, xend = 0))                  # wykres po prawej stronie
```

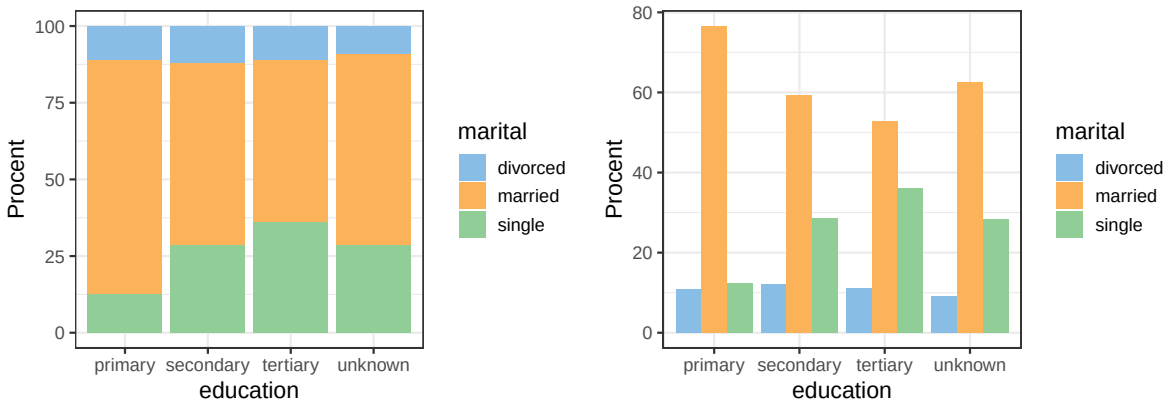
Powinienem jeszcze omówić sytuację, w której dodajemy kolejną zmienną kategorialną i umieszczamy ją w tzw. legendzie. Modyfikacje poprzednich przykładów będą niewielkie. Pierwsza z nich jest dość oczywista, bo wymaga od nas dołączenia kolejnego elementu estetycznego. Jeśli poziomy zmiennej będziemy różnicować wypełnieniem słupka, wtedy w `aes` zapiszemy `fill = nazwa_zmiennej`. Druga modyfikacja ma charakter preferencji wizualnych. Jeżeli chcemy otrzymać wykres słupkowy zestawiony (*stack*), w którym słupki umieszczane są jeden na drugim, wtedy nic nie musimy zmieniać. Jest to domyślne zachowanie funkcji. Jeśli jednak chcemy otrzymać wykres słupkowy zgrupowany, gdzie słupki pojawią się obok siebie, wtedy takie zachowanie wymusimy pisząc: `geom_col(position = "dodge")`.

W przykładzie ilustrującym pokażę, jak rozkładają się wartości procentowe dla zmiennej stan cywilny (`marital`) w zależności do wykształcenia (`education`). Konstruując tabelę, stanowiącą dane wejściowe dla wykresu, przyjmuję, że to kategorie zmiennej stan cywilny sumują się do 100%.

```
> eduMarit <- bank %>%
+    group_by(education, marital) %>%
+    summarise(Liczebnosc = n()) %>%
+    mutate(Procent = 100*Liczebnosc/sum(Liczebnosc))
> head(eduMarit, 3)
# A tibble: 3 x 4
# Groups:   education [1]
  education marital  Liczebnosc Procent
  <chr>     <chr>         <int>   <dbl>
1 primary   divorced        752    11.0
2 primary   married        5246    76.6
3 primary   single          853    12.5
```

Teraz możemy zbudować wykresy i to w dwóch wariantach.

```
> p6 <- ggplot(eduMarit, aes(x = education, y = Procent, fill = marital))
> p6 + geom_col()                    # wykres po lewej stronie; domyślnie jest position = "stack"
> p6 + geom_col(position = "dodge") # wykres po prawej stronie
```



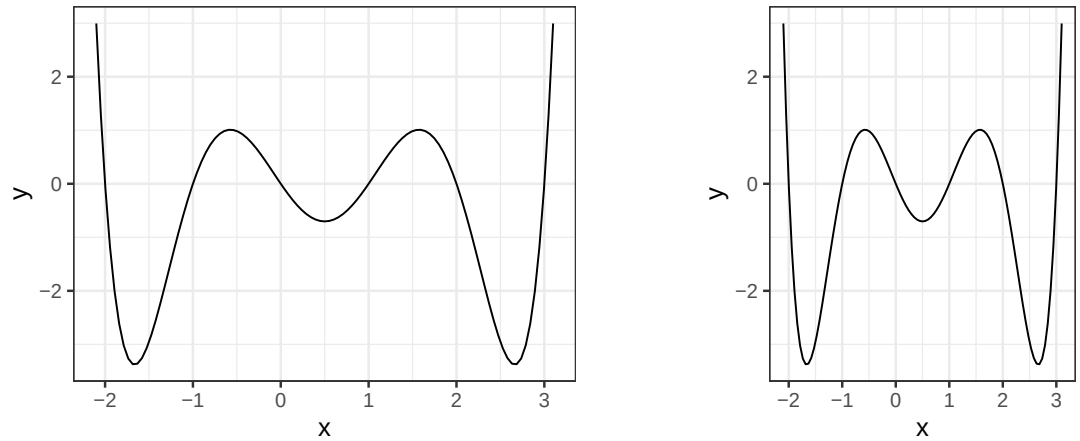
6.2.3. Wykresy rozkładu zmiennej. Rozkład zmiennej możemy przedstawić w postaci histogramu, funkcji gęstości estymatora jądrowego, dystrybuanty empirycznej, wykresu pudełko-wąsy i kwantyl-kwantyl (zob. tab. 6.1). Te wykresy budujemy dla zmiennych, których mamy realizację — dane. W R mamy też możliwości narysowania wykresu dowolnej funkcji zadanej równaniem. Od tego zaczniemy.

Wykresy funkcji

Czasami chcemy narysować rozkład teoretyczny i porównać go z empirycznym. W tym wypadku użyjemy funkcji `stat_function()`. Za jej pomocą narysujemy wykres dowolnej krzywej. Możemy wybrać funkcję już zaimplementowaną w R (np.: `sin()`, `log()`, `dnorm()` itp.) lub też samemu ją stworzyć. Zacznijmy od tej drugiej możliwości.

Naszym zadaniem jest narysowanie wykresu wielomianu 6 stopnia, którego postać zapiszemy w obiekcie o nazwie `Wielom6`. 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 przedział $(-2.1, 3.1)$. W poniższym przykładzie zwróć 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)`.

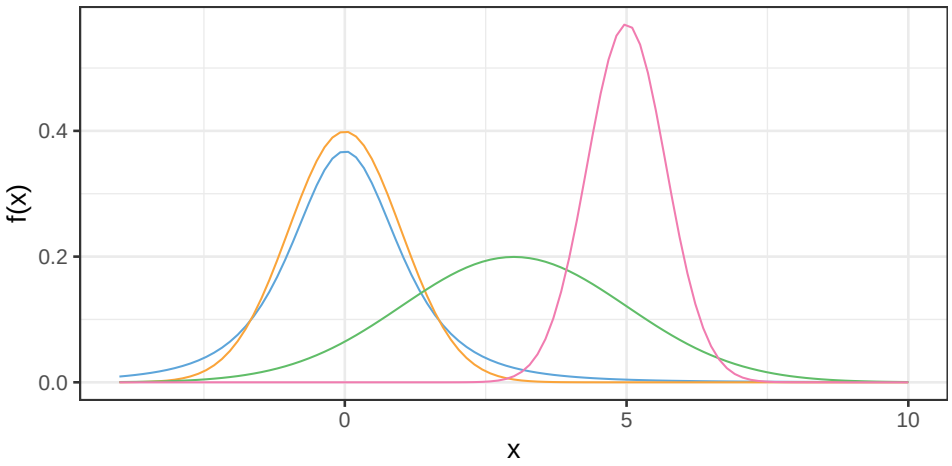
```
> ## Rysowanie wielomianu, o zadanie postaci funkcyjnej
> Wielom6 <- function(x) 0.2*x^6 - 0.6*x^5 - x^4 + 3*x^3 + 0.8*x^2 - 2.4*x
> wyk0 <- ggplot(data.frame(x = c(-2.1, 3.1)), aes(x = x))
> wyk <- wyk0 + stat_function(fun = Wielom6)
> wyk # odcinek [0,1] na osi X jest dłuższy od identycznego na osi Y
> wyk + coord_fixed(ratio = 1) #gwarancja, że długości będą identyczne
```



Przejdźmy teraz do narysowania kilku funkcji gęstości na tym samym wykresie. Postępowanie zasadniczo nie różni się od zaprezentowanego powyżej. Jednak zmuszeni jesteśmy do uwzględnienia argumentu opcjonalnego funkcji `stat_function()`, jeśli chcemy zmienić domyślne parametry rysowanych gęstości.

Narysujemy funkcję gęstość rozkładu t-Studenta. Wykorzystamy funkcję gęstości `dt()`, przyjmując liczbę 3 stopnie swobody (zob. str. 36). Następnie narysujemy trzy funkcje gęstości rozkładu normalnego za pomocą `dnorm()` (zob. str. 35). Za średnie przyjmiemy wartości: 0,3,5, a za odchylenia standardowe: 1,2,0.7. Taką możliwość daje nam argument opcjonalny `arg=list(..)`, w którym zamiast kropek podajemy nazwy parametrów i ich wartości. To znakomity moment, abyśmy użyli argumentu opcjonalnego `colour`, pozwalającego zmienić kolor linii.

```
> ## Rysowanie różnych funkcji gęstości
> dfX <- data.frame(osX = c(-4, 10)) #przedział zmienności x
> mdf <- ggplot(dfX, aes(x = osX)) + xlab("x") + ylab("f(x)")
> mdf + stat_function(fun = dt, colour = "#5DA5DA", args = list(df = 3)) +
+       stat_function(fun = dnorm, colour = "#FAA43A") + # normalny N(0,1)
+       stat_function(fun = dnorm, colour = "#60BD68", args = list(mean = 3, sd = 2)) +
+       stat_function(fun = dnorm, colour = "#F17CB0", args = list(mean = 5, sd = 0.7))
```



Histogram

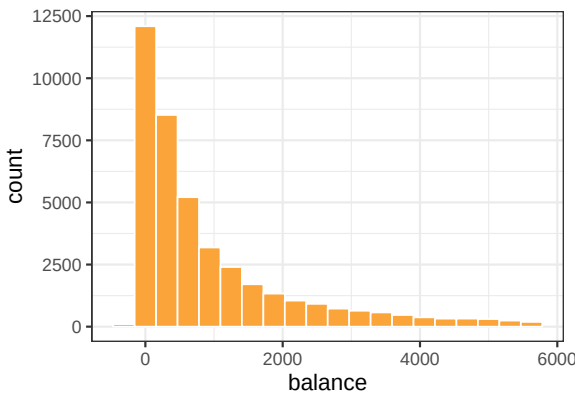
Prezentację rozkładów empirycznych rozpoczniemy od histogramu i funkcji `geom_histogram()`, która pojawiła się w tabeli 6.1. Domyślna wartość argumentu `bins` jest równa 30. Jeżeli chcesz tę liczbę przedziałów zmienić, wpisz po prostu swoją wartość — poeksperymentuj.

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])
```

```
> p7 <- ggplot(bankCut, aes(x = balance))
> p7  + geom_histogram(bins = 20)
```



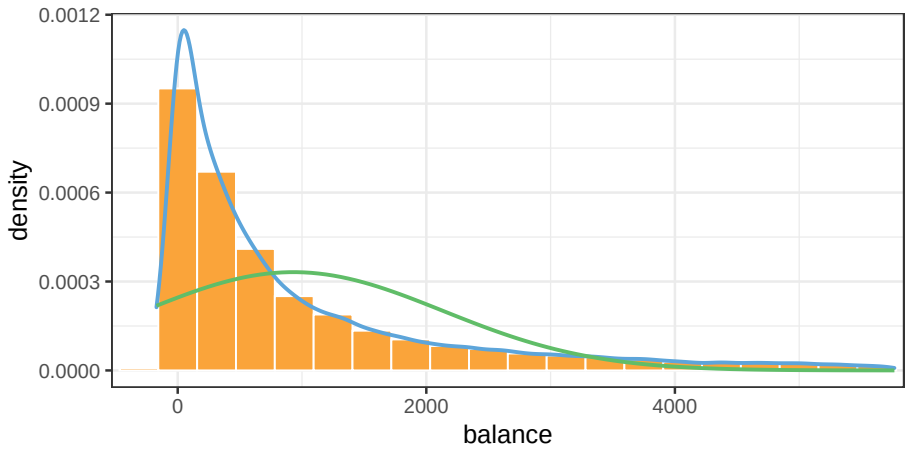
Strefa Eksperta 6.1

Dodawanie funkcji gęstości do histogramu

Dla histogramu przyjmowane są liczebności na osi rzędnej (Y). Jeśli zechcemy na histogram nanieść jakąkolwiek funkcję gęstości, 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..)`.

Poniżej ten sam przykład z naniesioną gęstością teoretyczną oraz gęstością estymatora jądrowego (o tym w kolejnym rozdziale). Przyjmijmy, że będzie to rozkład normalny ze średnią 930 i odchyleniem standardowym 1203 — wartości obliczone z próby. Jak widzisz, rozkład empiryczny w ogóle nie przypomina rozkładu normalnego (kolor zielony).

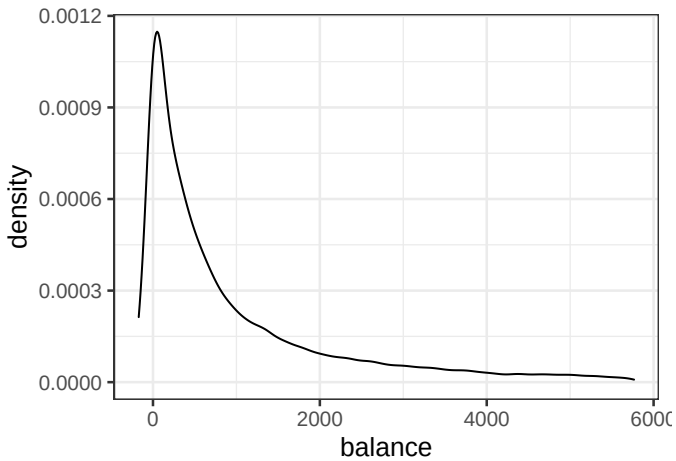
```
> p8 <- ggplot(bankCut, aes(x = balance))
> p8  + geom_histogram(aes(y = ..density..), fill = "#FAA43A", colour = "white", bins = 20) +
+       geom_density(colour = "#5DA5DA", size = 1) +
+       stat_function(fun = dnorm,
+                     args = list(mean = 930, sd = 1203),
+                     colour = "#60BD68", size = 1) +
+       scale_x_continuous(expand = c(0.01, 0.01))
```



Estymator gęstości jądrowej

Funkcja gęstości estymatora jądrowego `geom_density()` pozwala dokładniej przedstawić rozkład empiryczny zmiennej. Wykorzystajmy uprzednio przygotowaną ramkę danych `bankCut` i zbudujmy taki wykres dla zmiennej saldo rachunku (`balance`).

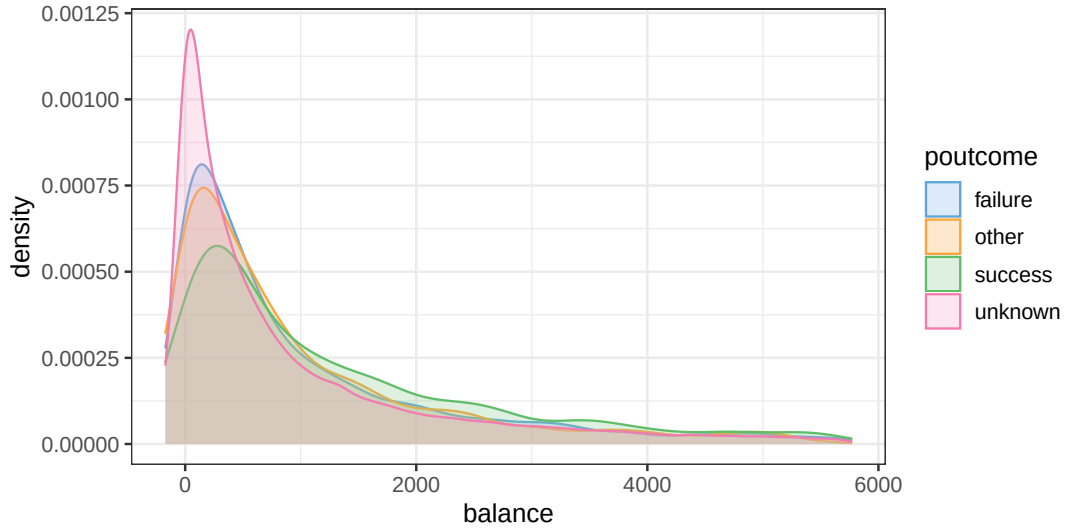
```
> p8 <- ggplot(bankCut, aes(x = balance))
> p8  + geom_density()
```



W dość prosty sposób możemy rozszerzyć powyższą składnię i nanieść kilka gęstości. Załóżmy, że chcemy zobaczyć, jak wyglądają gęstości ze względu na wynik ostatniej kampanii marketingowej (`poutcome`).

Jeśli chcemy wypełnić gęstości kolorem, wtedy potrzebujemy dodatkowego elementu estetycznego `fill`, który będzie wskazywał na zmienną `poutcome`. W tej sytuacji warto zmienić intensywność wypełnienia kolorem, sterując argumentem opcjonalnym $\alpha \in (0,1)$, np. `geom_density(alpha = 0.4)`. Mamy też możliwość zmiany koloru linii samej krzywej, jeżeli użyjemy kolejnego elementu estetycznego `colour`. Przyjmijmy, że kolory wypełnienia i linii będą identyczne. Poeksperymentuj z poniższym kodem, usuwając jeden z elementów estetycznych.

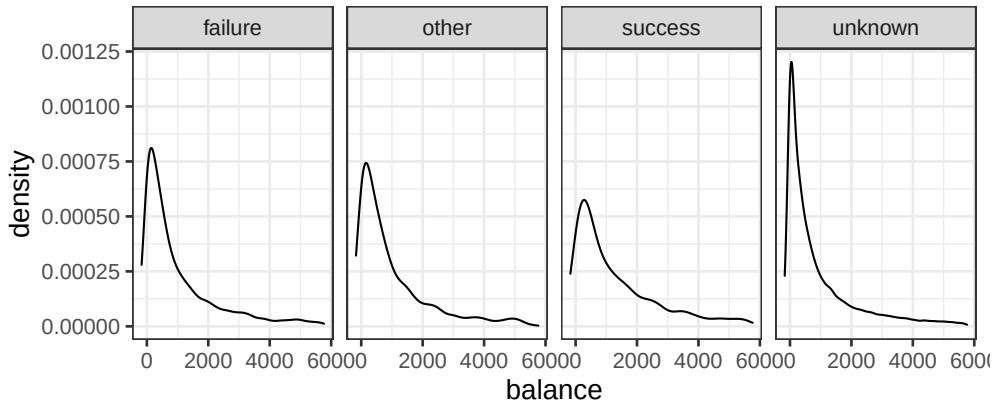
```
> p9 <- ggplot(bankCut, aes(x = balance, fill = poutcome, colour = poutcome))
> p9  + geom_density(alpha = 0.3)
```



WARTO WIEDZIEĆ

Zobacz, jak łatwo utworzyć alternatywną grafikę z wykorzystaniem paneli.

```
> ggplot(bankCut, aes(x = balance)) + facet_wrap(~ poutcome, nrow = 1) +
+                                     geom_density()
```

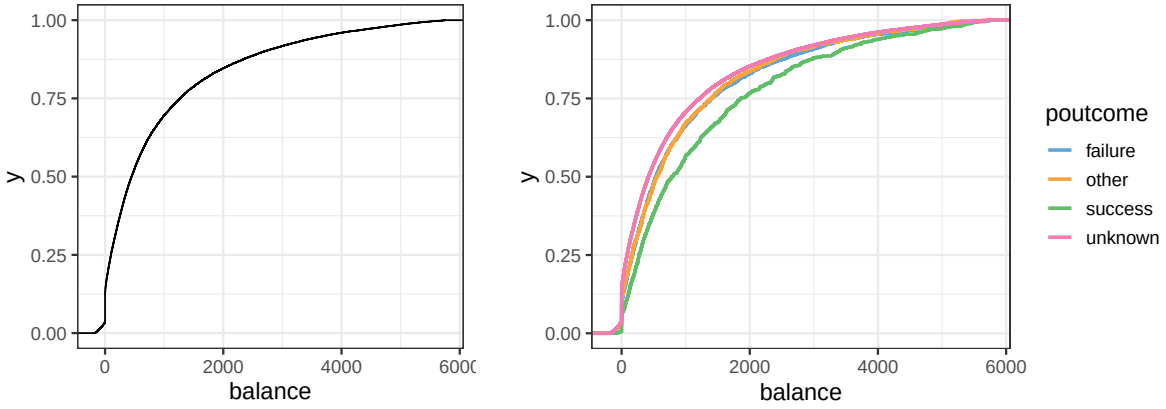


Dystrybuanta empiryczna

Dystrybuantę empiryczną narysujemy przy użyciu funkcji `stat_ecdf()`. Ostatni jej człon to skrót angielskiej nazwy: *empirical cumulative distribution 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: kolor czy rodzaj linii. W zależności od decyzji, wybierzemy jeden z elementów estetycznych: `colour` lub `linetype`. Dopuszczalne jest użycie obu naraz — sprawdź. Poniżej zamieszczam dwa przykłady.

```
> ggplot(bankCut, aes(x = balance)) +
+ stat_ecdf()
```

```
> ggplot(bankCut, aes(x = balance, colour = poutcome)) +
+ stat_ecdf()
```

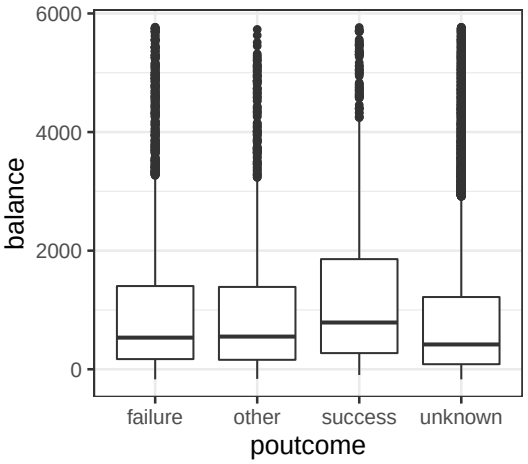


Wykres pudełko-wąsy

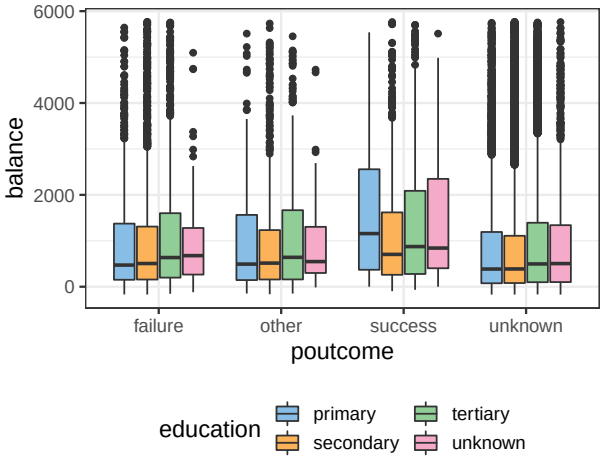
Kolejną funkcją pozwalającą ocenić rozkład zmiennej, jest `geom_boxplot()`. Rysuje ona wykres pudełko-wąsy wykorzystując statystyki pozycyjne. W definicji `ggplot()` oś X reprezentowana jest przez zmienną kategorialną, natomiast oś Y przez zmienną ciągłą — nie odwrotnie. Jeśli mamy życzenie zamienić osie, wtedy użyjmy wspomnianej wcześniej funkcji `coord_flip()`.

Kontynuujemy badanie rozkładu zmiennej saldo rachunku w zależności od wyników ostatniej kampanii marketingowej. Na drugim wykresie uwzględnimy jeszcze zmienną odnoszącą sie do poziomu edukacji (`education`). W ramach ćwiczenia zbuduj wykres, na którym w panelach pojawią się poziomy edukacji. Pamiętaj, że w tej sytuacji, będziemy mieć jeden element estetyczny mniej.

```
> ggplot(bankCut, aes(x = poutcome, y = balance)) +
+ geom_boxplot()
```




```
> ggplot(bankCut, aes(x = poutcome, y = balance,
+                     fill = education)) + geom_boxplot()
```



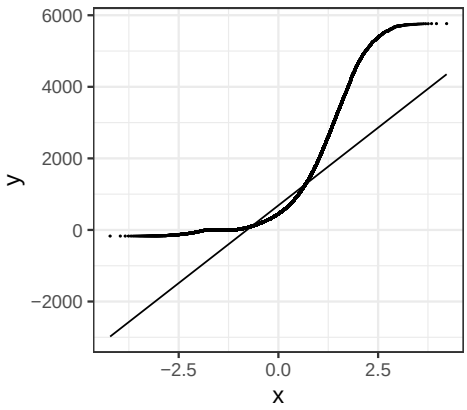
Wykres kwantyl-kwantyl

Aby sprawdzić, jak bardzo rozkład z próby jest podobny do wybranego rozkładu teoretycznego, możemy posłużyć się wykresem kwantyl-kwanty i funkcją `geom_qq()`. Swą nazwę zawdzięcza budowie: na osi X umieszczone są kwantyle teoretyczne, a na osi Y kwantyle empiryczne. Im większe podobieństwo między kwantylami, tym bardziej rozkład empiryczny jest zbliżony do rozkładu teoretycznego. W tej ocenie pomoże nam funkcja `geom_qq_line()`, która nanosi prostą na taki wykres. Idealne podobieństwo zaobserwujemy wtedy, gdy punkty będą leżały na prostej.

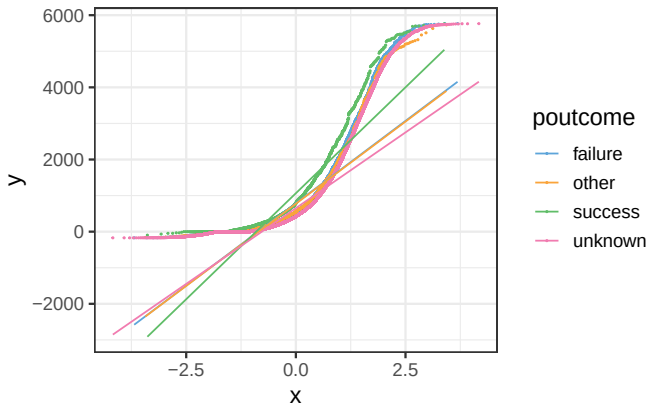
Jako rozkład teoretyczny możesz wybrać dowolny rozkład. W praktyce, najczęściej porównujemy rozkład naszej zmiennej z rozkładem normalnym. Dlatego domyślnym argumentem obu funkcji są kwantyle rozkładu normalnego `qnorm`. Poprzednie wykresy pokazały, że rozkład salda rachunku

znacznie odbiega od rozkładu normalnego. To powinno być bardzo widoczne na wykresie. Oprócz rozkładu samego salda, zbadamy jego rozkład w zależności od kampanii marketingowej.

```
> ggplot(bankCut, aes(sample = balance)) +
+ geom_qq() + geom_qq_line()
```




```
> ggplot(bankCut, aes(sample = balance, colour = poutcome)) +
+ geom_qq() + geom_qq_line()
```



6.3. Zadania

Zad. 1. Porównaj kształt funkcji gęstości rozkładu normalnego i rozkładu t-studenta, dla różnych stopni swobody, np. 3, 8, 20. W tym celu narysuj na jednym wykresie odpowiednie funkcje gęstości.

Zad. 2. Wykorzystaj zbiór danych `WholesaleCustomers.txt` i oblicz korelacje między wydatkami na produkty mleczne a pozostałymi produktami. Wartości tych korelacji wizualizować. Wersja trudniejsza: w analizie uwzględnić regiony. Wyniki zinterpretować.

Zad. 3. Dla każdego zadania z rozdziału 5 utwórz wykresy. Chodzi o zadania końcowe zaczynające się na stronie 66.

ROZDZIAŁ 7

Estymacja i testowanie hipotez

7.1. Estymacja przedziałowa średniej μ i proporcji p

Do zbudowania przedziału ufności dla średniej wykorzystamy następującą funkcję:

```
t.test(x, conf.level = 0.95)
```

Ma ona dwa argumenty. Pierwszy `x` — jest wektorem obserwacji, drugi `conf.level` — odnosi się do poziomu ufności. O tej funkcji powiem więcej, gdy przejdziemy do testowania hipotez statystyczny. Jedyną informację na jakiej musisz sie skupić to przedział ufności, który może zaczynać się następująco: 95 percent confidence interval. Zobacz poniższy kod.

```
> wydatki <- c(30, 25, 16, 22, 34, 54, 23, 28, 23, 21)
> t.test(wydatki, conf.level = 0.9)

One Sample t-test

data:  wydatki
t = 8, df = 9, p-value = 2e-05
alternative hypothesis: true mean is not equal to 0
90 percent confidence interval:
 21.5 33.7
sample estimates:
mean of x
     27.6
```

W wypadku estymacji proporcji użyjemy pakietu `Hmisc`, w którym funkcja:

```
binconf(x, n, alpha=0.05, method=c("wilson", "exact", "asymptotic", "all"))
```

pozwala oszacować przedział w sposób dokładny (`exact`) lub go aproksymować. Jeżeli próba jest duża, wtedy można założyć, że statystyka na bazie której taki przedział jest konstruowany, ma rozkład normalny (`asymptotic`). Przy niewielkich próbach najlepiej użyć metody Wilsona (nawet bardziej preferowana niż szacowanie dokładne). Wymagane argumenty funkcji to: `x` – liczba sukcesów, `n` – liczba obserwacji.

Zobaczmy, jak wyglądają oszacowania przedziałów w zależności od rozmiaru próby i wykorzystanej metody estymacji.

```
> ## gdy pakiet niezainstalowany użyj: install.packages("Hmisc")
> library(Hmisc)
> binconf(150, 300,  alpha = 0.05, method = "all") # różnice marginalne
           PointEst Lower Upper
Exact           0.5 0.442 0.558
Wilson          0.5 0.444 0.556
Asymptotic      0.5 0.443 0.557

> binconf(20, 40,  alpha = 0.05, method = "all") # większe różnice, bo próba mała
           PointEst Lower Upper
Exact           0.5 0.338 0.662
Wilson          0.5 0.352 0.648
Asymptotic      0.5 0.345 0.655
```

7.2. Testowanie hipotez

7.2.1. Testy dla frakcji. Do weryfikacji hipotezy o frakcji w jednej populacji lub równości frakcji w wielu populacjach wykorzystamy następującą funkcję:

```
prop.test(x, n, p = NULL, alternative = c("two.sided", "less", "greater"),
          conf.level = 0.95, correct = TRUE)
```

w której:

- `x` – liczba sukcesów (gdy jest jedna populacja) lub wektor liczby sukcesów (przynajmniej dwie populacje); alternatywnie możesz podać tabelę, przy czym pierwsza kolumna odnosi się do sukcesu, a druga do porażki (kolejność jest ważna);
- `n` – liczba prób (rozmiar próby); nie podajesz, jeżeli wcześniejszym argumentem jest tabela;
- `p` – weryfikowane prawdopodobieństwo/frakcja sukcesów; gdy nie podasz, wtedy domyślnie przyjmowane jest 0.5;
- `correct` – czy poprawka Yatesa na ciągłość ma być uwzględniona.

Wykorzystamy powyższą funkcję i zbiór danych `satysfakcja.dat`, aby zweryfikować dwie hipotezy.

Hipoteza 1: Frakcja osób wierzących w życie po śmierci (`wiaraZyciePo`) jest równa 3/4 (0.75).

Hipoteza 2: Frakcja osób niewietrzących w życie po śmierci (`wiaraZyciePo`) jest taka sama w każdej grupie wykształcenia (`edukacja`). Innymi słowy grupy odnoszące się do poziomu wykształcenia są jednorodne pod względem braku wiary w życie po śmierci.

```
> ## Hipoteza 1
> saty <- read.table("dane/satysfakcja.dat", header=TRUE, sep="\t")
> table(saty$wiaraZyciePo)

Nie Tak
236 974

> prop.test(974, 236+974, p=0.75)

1-sample proportions test with continuity correction

data:  974 out of 236 + 974, null probability 0.75
X-squared = 19, df = 1, p-value = 1e-05
alternative hypothesis: true p is not equal to 0.75
95 percent confidence interval:
 0.78 0.83
sample estimates:
  p
0.8

>
> ## Hipoteza 2
> (tab <- table(saty$edukacja, saty$wiaraZyciePo))

                Nie Tak
  Licencjat      29 164
  Magisterium    33  76
  Policealna     13  67
  Szkola srednia 122 529
  Szkola zawodowa  39 138

> ## Zobaczmy jak wygląda rozkład frakcji
> prop.table(tab, margin = 1) # procent sumuje się do 100 w wierszach
```

```
                 Nie  Tak
  Licencjat      0.15 0.85
  Magisterium    0.30 0.70
  Policealna     0.16 0.84
  Szkola srednia 0.19 0.81
  Szkola zawodowa 0.22 0.78

> prop.test(tab) #weryfikacja drugiej hipotezy

5-sample test for equality of proportions without continuity correction

data:  tab
X-squared = 12, df = 4, p-value = 0.02
alternative hypothesis: two.sided
sample estimates:
prop 1 prop 2 prop 3 prop 4 prop 5
  0.15   0.30   0.16   0.19   0.22
```

7.2.2. Testy niezależności χ^2. Weryfikację hipotezy o niezależności dwóch zmiennych kategorialny przeprowadzamy za pomocą testu χ^2 Pearsona, który został zaimplementowany w funkcji `chisq.test()`. Wymaganym argumentem tej funkcji jest tabela utworzona z dwóch zmiennych, w której komórki są liczebnościami.

Załóżmy, że chcemy zweryfikować hipotezę, że poziom deklarowanego szczęścia (`szczescie`) oraz wykształcenie (`edukacja`) są niezależne. Wykorzystujemy zbiór danych `satysfakcja.dat` oraz interesujące nas zmienne, aby zapisać:

```
> (tab <- table(saty$edukacja, saty$szczescie))

                 Bardzo szczesliwy Dosc szczesliwy Nieszczesliwy
  Licencjat                     79             100            14
  Magisterium                   30              67            12
  Policealna                    32              39             9
  Szkola srednia               185             381            85
  Szkola zawodowa               42             102            33

> chisq.test(tab)

Pearson's Chi-squared test

data:  tab
X-squared = 26, df = 8, p-value = 0.001
```

Jeśli sądzimy, że asymptotyczny rozkład statystyki testowej odbiega od rozkładu χ^2, wtedy możemy wykorzystać procedurę symulacji do oszacowania *p-value*. Wystarczy, że użyjemy dodatkowych argumentów funkcji `chisq.test()`. Ilustruję to przykładem.

```
> chisq.test(tab, simulate.p.value = TRUE, B = 2000) # B -ile powtórzeń w Monte Carlo

Pearson's Chi-squared test with simulated p-value (based on 2000 replicates)

data:  tab
X-squared = 26, df = NA, p-value = 0.001
```

7.2.3. Testy zgodności z rozkładem normalnym. W estymacji przedziałowej czy weryfikacji niektórych hipotez statystycznych przyjmujemy założenie, że rozkład badanej cechy jest normalny. Do weryfikacji takiego założenia możemy wykorzystać dostępną w pakiecie podstawowym `stats` funkcję `shapiro.test()`, która odnosi się do testu Shapiro-Wilka. Jeśli zdecydujemy się na pakiet `nortest`, wtedy mamy do dyspozycji dodatkowych 5 testów: `ad.test()` (Anderson-Darling), `cvm.test()` (Cramer-von Mises), `lillie.test()` (Kolmogorov-Smirnov z poprawką Lillieforsa), `pearson.test()` (Pearson χ^2,

`sf.test()` (Shapiro-Francia). Argumentem wejściowym we wszystkich funkcjach jest wektor, którego wartości odpowiadają badanej zmiennej.

Zweryfikujmy hipotezę, że liczba godzin pracy (`ileGodzPracuje`) ma rozkład normalny. Hipoteza alternatywna głosi, że zmienna ma rozkład inny niż rozkład normalny. Dla porównania użyjemy dwóch testów.

```
> ## Jeżeli pakiet nie jest zainstalowany, to użyj: install.packages("nortest")
> library(nortest)
> shapiro.test(saty$ileGodzPracuje) # Shapiro-Wilk

Shapiro-Wilk normality test

data:  saty$ileGodzPracuje
W = 0.9, p-value <2e-16

> ad.test(saty$ileGodzPracuje) # Anderson-Darling

Anderson-Darling normality test

data:  saty$ileGodzPracuje
A = 18, p-value <2e-16
```

7.2.4. Testy dla wartości średniej lub mediany. Hipotezę o wartości średniej weryfikujemy w jednej, dwóch lub wielu populacjach. Pierwsza możliwość zakłada, że średnia jest równa zadanej wartości. Druga grupa hipotez odnosi się do testowania równości wartości średnich w dwóch niezależnych próbach[1]. Jeżeli obserwacje zestawione są w pary, wtedy mówimy, że próby są zależne, a weryfikowana hipoteza zakłada, że różnica między wartościami średnimi jest równa zadanej wartości (najczęściej to zero). Te trzy warianty hipotez weryfikujemy tą samą funkcją – zmieniamy, w zależności od potrzeby, jej argumenty:

```
t.test(x, y = NULL, alternative = c("two.sided", "less", "greater"),
       mu = 0, paired = FALSE, var.equal = FALSE, conf.level = 0.95)
```

gdzie:

- `x,y` – wartości numeryczne zmiennej lub zmiennych
- `alternative` – test dwustronny (`two.sided`), jednostronny dla alternatywnej hipotezy (`less` - lewostronny; `greater` - prawostronny);
- `mu` – założenie dotyczące wartości średniej;
- `paired` – czy obserwacje są zestawione w pary;
- `var.equal` – czy założono równość wariancji; w wypadku braku równości zostanie wykorzystana korekta Welcha.

W poniższych przykładach będę wykorzystywał zbiór danych `satysfakcja.dat`. Zwracam uwagę na dwie sprawy. Pierwsza – przy weryfikacji hipotez rezygnuję ze sprawdzania, czy badana cecha lub cechy mają rozkład normalny. Przykłady ilustrują tylko wykorzystanie funkcji `t.test()`. Druga – jeśli nie wspomnę o poziomie istotności, to zakładam, że $\alpha = 0.05$.

Zweryfikujmy hipotezę, że liczba godzin pracy (`ileGodzPracuje`) jest różna od 8 godzin. Stawiamy hipotezę zerową i alternatywną: $H_0: \mu = 8$ i $H_1: \mu \neq 8$.

```
> ## H_1: m =/= 8
> t.test(saty$ileGodzPracuje, mu=8, alternative = "two.sided")

One Sample t-test

data:  saty$ileGodzPracuje
t = -12, df = 1209, p-value <2e-16
alternative hypothesis: true mean is not equal to 8
95 percent confidence interval:
```

[1] Rozszerzeniem, na przypadek wielu prób, jest analiza wariancji ANOVA.

```
 6.5 6.9
sample estimates:
mean of x
      6.7
```

Oszacowana średnia liczba godzin pracy wyniosła: 6.69. Ponieważ p-wartość = 0, więc hipotezę H_0 odrzucamy na korzyść hipotezy alternatywnej (przy uwzględnieniu $\alpha = 0.05$). Daje nam to podstawę do twierdzenia, że liczba godzin pracy jest różna od 8.

Interesujące może być również pytanie, czy przeciętna liczba godzin pracy jest taka sama dla kobiet i mężczyzn (`plec`). Prowadzi nas ono do następujących hipotez: $H_0: \mu_k = \mu_M$ i $H_1: \mu_k \neq \mu_M$, gdzie indeks odnosi się do kobiet bądź mężczyzn. W tym wypadku mamy do czynienia z dwoma niezależnymi próbami. Argumentami wejściowymi funkcji `t.test()` mogłyby być wektory `x` i `y` reprezentujące odpowiednio liczbę godzin pracy mężczyzn i kobiet. Biorąc pod uwagę konieczność przygotowania takich wektorów (na podstawie ramki danych), wygodniej jest użyć formuły. Zobacz poniżej.

```
> ## ogólna formuła: zmiennaIlościowa ~ zmiennaKategorialna
> t.test(ileGodzPracuje ~ plec, data=saty) # data= nazwaZbioruDanych

Welch Two Sample t-test

data:  ileGodzPracuje by plec
t = -0.2, df = 1182, p-value = 0.8
alternative hypothesis: true difference in means between group Kobieta and group Mezczyzna is not equal to 0
95 percent confidence interval:
 -0.47  0.38
sample estimates:
  mean in group Kobieta mean in group Mezczyzna
                    6.7                     6.7
```

Zwróćmy uwagę na przyjęte *implicite* założenie, że wariancje w obu grupach nie są jednakowe. Świadczy o tym wykorzystany tzw. test Welcha (korekta na `df`). Jeśli chcemy z tej korekty zrezygnować, powinniśmy hipotezę o równości wariancji w obu grupach zweryfikować, czyli $H_0: \sigma_K^2 = \sigma_M^2$ wobec $H_1: \sigma_K^2 \neq \sigma_M^2$. W sprawdzeniu tej hipotezy pomocna będzie funkcja `var.test()`, której odpowiada test F oparty na statystyce F-Snedecora. Również i tutaj musimy sprawdzić założenia jego stosowalności (jest nim rozkład normalny cech).

```
> ## Krok 1: hipoteza o równości wariancji
> var.test(ileGodzPracuje ~ plec, data=saty)

F test to compare two variances

data:  ileGodzPracuje by plec
F = 1, num df = 561, denom df = 647, p-value = 0.9
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
 0.86 1.19
sample estimates:
ratio of variances
                 1
```

Powyższy test nie daje podstaw do odrzucenia hipotezy zerowej, więc w kolejnym kroku zbadamy równość średnich z pominięciem korekty (zakładamy równość wariancji).

```
>  ## Krok 2: hipoteza o równości średnich, ale bez korekty
>  t.test(ileGodzPracuje ~ plec, data=saty, var.equal=TRUE)

Two Sample t-test

data:  ileGodzPracuje by plec
t = -0.2, df = 1208, p-value = 0.8
alternative hypothesis: true difference in means between group Kobieta and group Mezczyzna is not equal to 0
95 percent confidence interval:
 -0.47  0.38
sample estimates:
```

```
  mean in group Kobieta mean in group Mezczyzna
                    6.7                     6.7
```

Gdy rozkład cechy nie spełnia założeń i musimy zrezygnować z testu `t.test()`, wtedy możemy posłużyć się testem nieparametrycznym Wilcoxona i funkcją `wilcox.test()`. Jej argumenty wejściowe prawie pokrywają sie z omówionymi argumentami funkcji `t.test()`. Zobacz na poniższy przykład.

```
> ## Testy nieparametryczne; alternatywa dla t.test()
> # Przykład 1: H0: m <= 8; H1: m > 8
> wilcox.test(saty$ileGodzPracuje, mu=8, alternative = "two.sided")

Wilcoxon signed rank test with continuity correction

data:  saty$ileGodzPracuje
V = 2e+05, p-value <2e-16
alternative hypothesis: true location is not equal to 8

> # Przykład 2: H0: m_k = m_M; H1: m_K =/= m_M
> wilcox.test(ileGodzPracuje ~ plec, data=saty)

Wilcoxon rank sum test with continuity correction

data:  ileGodzPracuje by plec
W = 2e+05, p-value = 0.8
alternative hypothesis: true location shift is not equal to 0
```

7.3. Zadania

Zad. 1. Dla zmiennej saldo rachunku (`balance`) ze zbioru danych `bankFull.csv` zbudować przedział ufności dla średniej. Wynik zinterpretować.

Zad. 2. Dla zmiennej mówiącej o nieregulowaniu należności (`default`) ze zbioru danych `bank` zbudować przedział ufności. Wynik zinterpretować.

Zad. 3. Dla wybranego zbioru danych z katalogu `dane` oraz wybranych zmiennych zweryfikować wszystkie omówione w tym rozdziale hipotezy statystyczne. Pamiętać o sformułowaniu hipotezy zerowej i alternatywnej, założeniach oraz interpretacji wyników.

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.

[11] Wickham, H. (2014), 'Tidy data', *Journal of Statistical Software, Articles* **59**(10), 1–23.
URL: *https://www.jstatsoft.org/v059/i10*

[12] Wilkinson, L. (2005), *The Grammar of Graphics*, Springer-Verlag, Berlin, Heidelberg.

Indeks funkcji