Przemysław Biecek

# Przewodnik po pakiecie

Oficyna Wydawnicza GiS

Wnioski, skargi i zażalenia należy kierować na adres
dr hab. inż. Przemysław Biecek
Interdyscyplinarne Centrum Modelowania Matematycznego i Komputerowego
Uniwersytet Warszawski
e-mail: przemyslaw.biecek@gmail.com
strona www: http://www.biecek.pl

Redakcja i korekta
Karolina Biecek, Krzysztof Banas (rozdziały 1 i 2), Krzysztof Trajkowski (rozdział 3)

Projekt okładki:
Jakub Rakusa-Suszczewski

Skład komputerowy książki w systemie LaTeX:
Przemysław Biecek

*Przygotowałem tę książkę, aby przedstawić możliwości programu R, filozofię pracy z tym programem, pokazać dobre praktyki dotyczące codziennej pracy z danymi w R. Liczę, że książka ta przyda się tym wszystkim, którzy w pracy lub w szkole zajmują się analizą danych. Książka może być wykorzystana, jako pomoc w nauce programu R lub jako atlas funkcjonalności i możliwości tego programu. W latach 2010 – 2013 używałem jej z powodzeniem jako główny podręcznik do przedmiotu „Programowanie w R". Mam nadzieje, że pomoże ona odkryć przyjemność w pracy z danymi i programem R.*
*Pod adresem* http://www.biecek.pl/R/ *czytelnik znajdzie dodatkowe informacje, rozwiązania zadań umieszczonych w tej książce, oraz odnośniki do innych materiałów pomagających w nauce programu R.*

Copyright © 2008, 2011, 2014 by Przemysław Biecek

Wszelkie prawa zastrzeżone. Żadna część niniejszej publikacji, zarówno w całości, jak i we fragmentach, nie może być reprodukowana w sposób elektroniczny, fotograficzny i inny bez pisemnej zgody posiadaczy praw autorskich.

Wydanie trzecie rozszerzone, Wrocław 2014
ISBN: 978-83-62780-22-8
**Oficyna Wydawnicza GiS, s.c.,** http://www.gis.wroc.pl
**Druk i oprawa: I-BiS Usługi Komputerowe Wydawnictwo s.c.**

# Spis treści

**Spis treści** **v**

**Zanim zaczniemy tę przygodę** **ix**

**1 Łagodne wprowadzenie do R** **1**
1.1 Jak korzystać z tej książki? . . . . . . . . . . . . . . . . . . . . 1
1.2 Słów kilka o projekcie R . . . . . . . . . . . . . . . . . . . . 2
1.3 Instalacja . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
1.4 RStudio, uniwersalny edytoR . . . . . . . . . . . . . . . . . . 7
1.5 Startujemy . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
1.5.1 Pierwsze uruchomienie . . . . . . . . . . . . . . . . . 9
1.5.2 Gdzie szukać pomocy? . . . . . . . . . . . . . . . . . 10
1.5.3 kalkuRator . . . . . . . . . . . . . . . . . . . . . . . 12
1.5.4 Kilka przykładowych sesji w programie R . . . . . . . . . 15
1.5.5 Podstawy składni języka R . . . . . . . . . . . . . . . . 20
1.5.6 Wyświetlanie i formatowanie obiektów . . . . . . . . . . 34
1.6 Przyspieszamy . . . . . . . . . . . . . . . . . . . . . . . . . 37
1.6.1 Instrukcje warunkowe i pętle . . . . . . . . . . . . . . . 37
1.6.2 Funkcje . . . . . . . . . . . . . . . . . . . . . . . . . 43
1.6.3 Leniwa ewaluacja . . . . . . . . . . . . . . . . . . . . 52
1.6.4 Zarządzanie obiektami w przestrzeni nazw . . . . . . . . . 54
1.6.5 Wprowadzenie do grafiki . . . . . . . . . . . . . . . . . 57
1.6.6 Wprowadzenie do operacji na plikach i katalogach . . . . 59
1.7 Zadania do części „Łagodne wprowadzenie...” . . . . . . . . . . . 63

**2 pazuRrry** **67**
2.1 Typy zmiennych i operacje na nich . . . . . . . . . . . . . . . . 67
2.1.1 Typ czynnikowy/wyliczeniowy . . . . . . . . . . . . . . 67
2.1.2 Typ znakowy . . . . . . . . . . . . . . . . . . . . . . 70
2.1.3 Wektory . . . . . . . . . . . . . . . . . . . . . . . . . 71
2.1.4 Listy . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
2.1.5 Ramki danych . . . . . . . . . . . . . . . . . . . . . . 75
2.1.6 Macierze . . . . . . . . . . . . . . . . . . . . . . . . . 78
2.1.7 Obiekty . . . . . . . . . . . . . . . . . . . . . . . . . 86
2.1.8 Klasy . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
2.1.9 Formuły . . . . . . . . . . . . . . . . . . . . . . . . . 91

2.2 Przetwarzanie potokowe . . . . . 93
2.2.1 Pakiety plyr i reshape2 . . . . . 94
2.2.2 Rodzina funkcji *apply . . . . . 98
2.3 Wybrane funkcje matematyczne . . . . . 101
2.3.1 Wielomiany . . . . . 101
2.3.2 Bazy wielomianów ortogonalnych . . . . . 102
2.3.3 Operacje na zbiorach . . . . . 104
2.3.4 Szukanie maksimum/minimum/zer funkcji . . . . . 104
2.3.5 Rachunek różniczkowo–całkowy . . . . . 105
2.4 Zapisywanie i odczytywanie danych . . . . . 107
2.4.1 Zapisywanie i odczytywanie danych z plików . . . . . 107
2.4.2 Zapisywanie i odczytywanie danych z baz danych . . . . . 117
2.4.3 Inne sposoby odczytywania i zapisywania danych . . . . . 119
2.4.4 Zapisywanie grafiki do pliku . . . . . 121
2.5 Tryb wsadowy . . . . . 123
2.6 Powtarzalne badania, automatyzowane raporty a programowanie objaśniające . . . . . 124
2.6.1 Pakiet `knitr` a raporty w języku markdown / HTML . . . 124
2.6.2 Pakiet `slidify` a prezentacje w HTML5 . . . . . 128
2.6.3 Pakiet `Sweave` a raporty w języku LaTeX . . . . . 130
2.7 Budowa aplikacji www z pakietem `shiny` . . . . . 136
2.8 Budowanie własnych pakietów . . . . . 141
2.8.1 Niezbędne oprogramowanie . . . . . 141
2.8.2 Przygotowanie pakietu . . . . . 143
2.8.3 Weryfikacja, budowanie i instalacja pakietu . . . . . 148
2.9 Debugger i profiler . . . . . 151
2.9.1 Debugger . . . . . 151
2.9.2 Profiler . . . . . 157
2.9.3 Jak zwiększyć wydajność programów w R . . . . . 160
2.9.4 Przydatne funkcje systemowe . . . . . 165
2.10 Zadania do części „pazuRrry” . . . . . 166

**3 Wybrane procedury statystyczne 169**
3.1 Statystyki opisowe . . . . . 170
3.1.1 Podstawowe liczbowe statystyki opisowe . . . . . 170
3.1.2 Podstawowe graficzne statystyki opisowe . . . . . 174
3.2 Generatory liczb losowych . . . . . 182
3.2.1 Wprowadzenie do generatorów liczb pseudolosowych . . 184
3.2.2 Popularne rozkłady zmiennych losowych . . . . . 186
3.2.3 Wybrane metody generowania zmiennych losowych . . . 192
3.2.4 Estymacja parametrów rozkładu . . . . . 202
3.3 Wstępne przetwarzanie danych . . . . . 203
3.3.1 Brakujące obserwacje . . . . . 203
3.3.2 Normalizacja, skalowanie i transformacje nieliniowe . . . 206
3.4 Analiza wariancji, regresja liniowa i logistyczna . . . . . 210
3.4.1 Wprowadzenie do analizy wariancji . . . . . 212

3.4.2 Analiza jednoczynnikowa . . . . . . . . . . . . . . . . . . . . 213
3.4.3 Analiza wieloczynnikowa . . . . . . . . . . . . . . . . . . . . 221
3.4.4 Regresja . . . . . . . . . . . . . . . . . . . . . . . . . . . 225
3.4.5 Wprowadzenie do regresji logistycznej . . . . . . . . . . . 240
3.5 Testowanie . . . . . . . . . . . . . . . . . . . . . . . . . . . . 256
3.5.1 Testowanie zgodności . . . . . . . . . . . . . . . . . . . . . 257
3.5.2 Testowanie hipotezy o równości parametrów położenia . . 264
3.5.3 Testowanie hipotezy o równości parametrów skali . . . . . 267
3.5.4 Testowanie hipotez dotyczących wskaźnika struktury . . . 270
3.5.5 Testy istotności zależności pomiędzy dwoma zmiennymi . 272
3.5.6 Testowanie zbioru hipotez . . . . . . . . . . . . . . . . . . 280
3.5.7 Symulacyjne i permutacyjne wyznaczanie rozkładu statystyki testowej . . . . . . . . . . . . . . . . . . . . . . . . . 283
3.6 Bootstrap . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 286
3.6.1 Ocena rozkładu estymatora oraz wyznaczanie przedziału ufności dla parametru . . . . . . . . . . . . . . . . . . . . . 287
3.6.2 Testy bootstrapowe . . . . . . . . . . . . . . . . . . . . . . 290
3.7 Analiza przeżycia . . . . . . . . . . . . . . . . . . . . . . . . . 292
3.7.1 Krzywa przeżycia . . . . . . . . . . . . . . . . . . . . . . . 292
3.7.2 Model Coxa . . . . . . . . . . . . . . . . . . . . . . . . . . 294
3.8 Zadania do części „Wybrane procedury statystyczne” . . . . . . . 298

**4 gRrrafika 301**
4.1 Pakiet graphics . . . . . . . . . . . . . . . . . . . . . . . . . . 302
4.1.1 Wykres paskowy . . . . . . . . . . . . . . . . . . . . . . . 302
4.1.2 Histogramy dwuwymiarowy i dla dwóch grup . . . . . . . 303
4.1.3 Wykres róża wiatrów . . . . . . . . . . . . . . . . . . . . . 304
4.1.4 Wykres słonecznikowy . . . . . . . . . . . . . . . . . . . . 304
4.1.5 Wykres kołowy . . . . . . . . . . . . . . . . . . . . . . . . 304
4.1.6 Wykres słupkowy . . . . . . . . . . . . . . . . . . . . . . . 306
4.1.7 Wykres kropkowy . . . . . . . . . . . . . . . . . . . . . . . 307
4.1.8 Wykres torbowy . . . . . . . . . . . . . . . . . . . . . . . . 307
4.1.9 Wykresy rozrzutu, dwu- i wielowymiarowe . . . . . . . . . 308
4.1.10 Wykres macierzy korelacji . . . . . . . . . . . . . . . . . . 310
4.1.11 Wykres konturowy dwuwymiarowego rozkładu normalnego 310
4.1.12 Wykres diagnostyczny . . . . . . . . . . . . . . . . . . . . . 310
4.1.13 Wykres koniczyny . . . . . . . . . . . . . . . . . . . . . . . 312
4.1.14 Wielowymiarowy, jądrowy estymator gęstości . . . . . . . 312
4.1.15 Wykresy konturowe . . . . . . . . . . . . . . . . . . . . . . 314
4.1.16 Wykres mapa ciepła . . . . . . . . . . . . . . . . . . . . . . 314
4.1.17 Wykres profili obserwacji . . . . . . . . . . . . . . . . . . . 316
4.1.18 Interaktywne wykresy z pakietem iplots . . . . . . . . . . 316
4.1.19 Wykres radarowy i twarze Chernoffa . . . . . . . . . . . . 318
4.2 Pakiet graphics - pełna kontrola . . . . . . . . . . . . . . . . . 319
4.2.1 Funkcja plot() . . . . . . . . . . . . . . . . . . . . . . . . . 320
4.2.2 Funkcja matplot() . . . . . . . . . . . . . . . . . . . . . . . 320

4.2.3 Osie wykresu . . . . . . . . . . . . . . . . . . . . . . . . . . . 321
4.2.4 Legenda wykresu . . . . . . . . . . . . . . . . . . . . . . . . 322
4.2.5 Funkcja image() . . . . . . . . . . . . . . . . . . . . . . . . . 323
4.2.6 Wyrażenia matematyczne . . . . . . . . . . . . . . . . . . . 324
4.2.7 Kolory . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 324
4.2.8 Właściwości linii . . . . . . . . . . . . . . . . . . . . . . . . 326
4.2.9 Właściwości punktów/symboli . . . . . . . . . . . . . . . . 327
4.2.10 Atomowe, niskopoziomowe funkcje graficzne . . . . . . . 328
4.2.11 Interaktywne odczytywanie wartości z wykresu . . . . . . 334
4.2.12 Tytuł, podtytuł i opisy osi wykresu . . . . . . . . . . . . . 334
4.2.13 Pozycjonowanie wykresu, wiele wykresów na rysunku . . 335
4.2.14 Parametry graficzne . . . . . . . . . . . . . . . . . . . . . . 336
4.3 Pakiet lattice . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 340
4.3.1 Wprowadzenie . . . . . . . . . . . . . . . . . . . . . . . . . 341
4.3.2 Szablony wykresów . . . . . . . . . . . . . . . . . . . . . . 342
4.3.3 Formuły i wybór zmiennych . . . . . . . . . . . . . . . . . 342
4.3.4 Panele i mechanizm warunkowania . . . . . . . . . . . . . 342
4.3.5 Mechanizm grupowania . . . . . . . . . . . . . . . . . . . . 343
4.3.6 Obiekty klasy trellis . . . . . . . . . . . . . . . . . . . . . . 343
4.3.7 Legenda wykresu . . . . . . . . . . . . . . . . . . . . . . . . 348
4.3.8 Pozycjonowanie wykresu, wiele wykresów na rysunku . . 348
4.3.9 Proporcje jednostek na osiach i reguła 45 stopni . . . . . . 350
4.3.10 Motywy i parametry graficzne . . . . . . . . . . . . . . . . 352
4.3.11 Więcej o panelach . . . . . . . . . . . . . . . . . . . . . . . 352
4.3.12 Atlas funkcji graficznych z pakietu `lattice` . . . . . . . . 356
4.3.13 Przykład dla badania „Diagnoza Społeczna" . . . . . . . . 366
4.4 Pakiet ggplot2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 369
4.4.1 Wprowadzenie . . . . . . . . . . . . . . . . . . . . . . . . . 369
4.4.2 Mapowania zmiennych na właściwości wykresu . . . . . . 371
4.4.3 Geometrie wykresów . . . . . . . . . . . . . . . . . . . . . . 372
4.4.4 Warstwy wykresów . . . . . . . . . . . . . . . . . . . . . . . 376
4.4.5 Mechanizm warunkowania . . . . . . . . . . . . . . . . . . 378
4.4.6 Motywy i kompozycje graficzne . . . . . . . . . . . . . . . 379
4.4.7 Układ współrzędnych i osie wykresu . . . . . . . . . . . . 380
4.4.8 Pozycjonowanie wykresu, wiele wykresów na rysunku . . 381
4.4.9 Agregaty i statystyki . . . . . . . . . . . . . . . . . . . . . . 382
4.4.10 Przykład dla badania „Diagnoza Społeczna" . . . . . . . . 382
4.5 Zadania do części „gRrrafika" . . . . . . . . . . . . . . . . . . . 385

**Opis zbiorów danych użytych w tej książce** **387**

**Bibliografia** **389**

**Skorowidz** **393**

# Zanim zaczniemy tę przygodę

## Przedmowa do wydania trzeciego

Pierwsze wydanie „Przewodnika..." zostało opracowane w latach 2006-2007, czyli ponad 7 lat temu. Od tego czasu program R bardzo się zmienił. Liczba dodatkowych bibliotek zwiększyła się wielokrotnie, przekraczając w tym roku cztery tysiące bibliotek. W międzyczasie rozwinął się również silnik R. Z wersji 2.6.0, przez wersję 2.15.1, aż do wersji 3.0.1 wydanej w kwietniu 2013 roku.

Równie duża zmiana dotyczy autora tego „Przewodnika...". Przez te kilka lat miałem przyjemność prowadzić wykłady, ćwiczenia i laboratoria dla studentów Politechniki Wrocławskiej i Warszawskiej, Uniwersytetu Warszawskiego i Wrocławskiego. Szkolenia dla pracowników naukowych, biologów czy genetyków oraz pracowników instytucji finansowych, banków czy firm konsultingowych. Miałem więc możliwość obserwowania, w jaki sposób różne osoby poznają R, co im ułatwia a co utrudnia naukę i pracę z tym programem.

Moja robocza teoria dotycząca efektywnego poznawania R wyróżnia trzy etapy nauki. Pierwszy etap można nazwać „zbieraniem motywacji". Początkujące osoby zauważają, że do nauki R trzeba włożyć trochę wysiłku i zastanawiają się czy warto. Aby ich przekonać, pręży się muskuły R, pokazuje jak można tworzyć aplikacje sieciowe, animacje, jak prosto zbudować wyrafinowany model statystyczny, jak prosto wygenerować raport czy powtórzyć obliczenia dla zmodyfikowanych danych. Ten etap najlepiej przejść słuchając prezentacji zaawansowanych użytkowników R, na żywo czy w formacie video np. na portalu `youtube.com`.

Drugi etap to poznawanie zasad funkcjonowania programu R, poznawanie techniki programowania w R, uczenie się filozofii pracy z tym pakietem, budowa mapy/atlasu pojęć związanych z programem R. Na tym etapie idealnie sprawdzają się podręczniki. Jeżeli nie są poświęcone opisowi wąskich, specjalistycznych rozwiązań to mogą zachować aktualność przez wiele lat. Tempo uczenia można sobie indywidualnie dostosować, niektóre rozdziały pominąć, inne przeczytać kilkukrotnie. Na marginesach umieścić komentarze i notatki. Mam nadzieję, że na tym etapie „Przewodnik po pakiecie R" będzie bardzo pomocny.

Trzeci etap to szlifowanie i uaktualnianie informacji o programie R. Pakiet się rozwija, warto poznawać nowe możliwości, czy też stare możliwości o których wcześniej nie słyszeliśmy. Tutaj idealnie sprawdza się czytanie blogów (pod adresem `http://www.r-bloggers.com/` dostępny jest agregator kilkudziesięciu blogów o R), uczestnictwo w konferencjach, nieformalnych spotkaniach użytkowników R, czy też prozaiczne googlowanie wokół interesujących nas tematów. Na tym

etapie bardzo pomocny jest dostęp do Internetu. Zarówno biblioteki programu R zmieniają się szybko, jak i szybko pojawiają się nowe bardziej efektywne, czasem efektowne, rozwiązania.

Na każdym z tych etapów pomocne jest pracowanie z innym użytkownikiem/użytkownikami R. Ponieważ większość problemów można rozwiązać na wiele sposobów, dlatego też bardzo dużo można nauczyć się porównując różne sposoby rozwiązania tego samego problemu. Ostatnio prowadząc zajęcia zadaję studentom ten sam projekt do wykonania, następnie najlepsze projekt są omawiane publicznie. To niesamowite, jak różne są te rozwiązania i jak wiele interesujących ciekawostek różne osoby wyszukują. A to efektywniejsze wczytywanie danych, a to zmiana kroju pisma na wykresie, a to animacja zanurzona w pliku pdf.

Jeżeli chodzi o takie ciekawostki czy nowości, nie sposób przecenić użyteczności Internetu i portali typu `http://stats.stackexchange.com` (pytania dotyczące statystyki) lub `http://stackoverflow.com/questions/tagged/r` (pytania dotyczące programowania). Głównie z powodu łatwości wyszukiwania informacji w Internecie. Wiedząc czego się szuka w kilka minut można znaleźć odpowiedź na jakimś forum, w dokumentacji czy roboczych notatkach innego użytkownika. Papierowa „mini encyklopedia" nie będzie nawet w części tak przydatna jak internetowa wyszukiwarka. Jednak uczenie się wyłącznie na bazie samych ciekawostek i zespołowych projektów (a i takie eksperymenty robiłem) owocuje fragmentaryczną wiedzą. Ciekawostki są otaczane koślawymi rozwiązaniami, czcionka na wykresie jest ciekawa ale sam wykres niedopracowany a dane błędnie przetworzone.

Dlatego też uważam, że elektroniczne wersje podręczników to niezbyt dobry pomysł. Wyszukiwanie w takich podręcznikach nie jest tak wygodne jak w Internecie. A traci się wygodę pracy z papierową książką.

Jest wiec w procesie nauki R miejsce dla podręczników. Na drugiem etapie uczenia zbudowanie bazy, szkieletu, przedstawienie krok po kroku filozofii pracy z programem R jest bardzo ważne. Pozwala spojrzeć szerzej co jest dostępne na horyzoncie możliwości. Ułatwia późniejsze wyszukiwanie informacji i uaktualnianie naszej wiedzy o programie R. Uważam wręcz, że dobry kurs programu R to połączenie odpowiedniego podręcznika do samodzielnego czytania i projektowej pracy z rzeczywistymi problemami. Wymaga to czasu i wysiłku, ale owocuje znacznie lepszymi efektami niż uczenie się od początku na samych przykładach.

„Przewodnik po pakiecie R" powstał po to by towarzyszyć w nauce, by przedstawiać atlas rozwiązań, by wskazywać mielizny i rafy. Mam nadzieję, że ta pozycja oferuje szerokie a jednocześnie jednolite spojrzenie na wiele rozwiązań dostępnych w R. Zarówno jeżeli chodzi o możliwości obliczeniowe, możliwości w modelowaniu statystycznym jak i możliwości w zakresie prezentacji danych. Starałem się zachować rosnący poziom trudności, przygotowując łatwe do zrozumienia i interesujące przykłady z komentarzami. Zostawiając też miejsce dla czytelnika do samodzielnego sprawdzenia „co się stanie".

Trzecie wydanie przeszło gruntowny lifting. Usunąłem informacje o pakietach, które już nie są używane (np. świetny pakiet `CoCo` został już usunięty z serwerów `CRAN`), dodałem informacje o aktualnie używanych rozwiązaniach, takich jak np. RStudio (rewelacyjny edytor do R). Wciąż uczę się też pisać. Przeformułowałem wiele zdań, przykładów czy komentarzy sprzed kilku lat. Mam wrażenie, że pewne rzeczy potrafię teraz przedstawić lepiej, zobaczymy co będę o tym myślał za kolejnych kilka lat. Pisanie bloga jest moim zdaniem świetną praktyką języka. Choć oczywiście zdaję sobie sprawę, że jeszcze wiele rzeczy do poprawienia.

Zmiany dotyczą również formy. Zmieniłem krój pisma na Minion Pro, zmieniłem sposób formatowania kodu R i sposób umieszczania komentarzy. Papierowe książki nie męczą tak wzroku jak aktywnie świecące ekrany. Autor ma pełną kontrolę nad typografią i projektem książki, może rozmieścić treść tak by łatwo było porównać napis z lewej strony z napisem na prawej stronie. Przyjemny w czytaniu krój pisma może pozytywnie nastawić czytelnika. To efekty, których nie doceniałem kilka lat temu, teraz jednak przekonany jestem, że są one ważne.

Jest też kilka zupełnie nowych podrozdziałów, które odzwierciedlają moje obecne zainteresowania. W wydaniu drugim rozbudowany został rozdział o grafice i wydajnym przetwarzaniu. W tym wydaniu dodałem opisy pakietów `knitr`, `slidify` i `shiny` pozwalających na wsparcie komunikacji pomiędzy statystykiem a końcowym odbiorcą. Więcej miejsca poświęciłem tematowi powtarzalnych analiz i obliczeń, łatwych do zweryfikowania i powtórzenia.

Powtarzalność badań odgrywa olbrzymią rolę. Zarówno w nauce jak i przemyśle. Jest to tak ważne zagadnienie, że powinien towarzyszyć kursom ze statystyki i analizy danych dla wszystkich studentów nauk medycznych, przyrodniczych i oczywiście technicznych. W przemyśle pozwala zwiększyć produktywność i oszczędzić dużo czasu, zarówno gdy wracamy do pewnego zagadnienia po pewnym czasie, jak i gdy musimy ponownie wykonać pewien raport. W nauce możliwość odtworzenia wyników to fundamentalny wymóg, każdego rzetelnego badania. Mam nadzieję, że ten podręcznik przedstawi możliwości R w zakresie badań powtarzalnych.

Na koniec chciałbym podziękować wszystkim osobom, które przyczyniły się do powstania tego wydania. Jest ich na szczęście zbyt wiele by wymienić je wszystkie. Najbardziej dziękuję mojej żonie, Karolinie, bez której nie tylko to wydanie ale też wiele innych projektów nigdy by nie powstało.

Moją aspiracją przy pisaniu trzeciego wydania było zbudowanie i pokazanie atlasu rozwiązań dostępnych w R, pozwalających na zrozumienie możliwości, logiki, schematów pracy z R. Programowanie w R może być przyjemnością, która wynika z braku ograniczeń jak i z estetyki dostępnych rozwiązań. Mam nadzieję, że uda mi się tę estetykę i tę przyjemność pracy z danymi pokazać. Życzę więc wielu fantastycznych wrażeń w pracy z programem R.

*Przemysław Biecek, Paryż 2013*

## Przedmowa do wydania drugiego

Minęły już ponad dwa lata od pierwszego wydania „Przewodnika ...". W międzyczasie program R rozwija się w wykładniczym tempie i zapewnił sobie pozycję wyśmienitego narzędzia do analizy danych. Liczba pakietów dostępnych w repozytorium CRAN zwiększyła się w ciągu ostatnich dwóch lat z 700 do 2400. Przybyło wiele funkcjonalności, pewne uległy zmianie, pewne rzeczy można zrobić już znacznie łatwiej. Widząc ilość zmian zacząłem się zastanawiać nad aktualizacją tego podręcznika.

Kolejna zachęta do zmian przyszła od czytelników od których przez te dwa lata otrzymałem wiele listów. Część listów ograniczających się do stwierdzenia „dobra

robota" część z sugestiami, co warto zmienić, co warto dodać, co lepiej usunąć. Były też listy z pogróżkami czy ultimatum, albo poprawię wskazany błąd na stronie *x* w linii *y* albo zostanę narażony na interwencję profesora Jana Miodka. Gorąco dziękuję za listy i uwagi! Miło jest wiedzieć, że ktoś poświęcił trochę czasu by podzielić się wrażeniami, przesłać kilka pomysłów, wskazać co mu się podoba, a co nie.

Część z propozycji rozszerzenia tej książki dotyczyło rozdziału dotyczącego statystyki i data mining. Zamiast jednak dodać kolejne sto stron do tej książki stwierdziłem, że właściwiej będzie napisać kolejną poświęconą wyłącznie analizie statystycznej danych w programie R z wykorzystaniem modeli liniowych i mieszanych. Prace nad tą pozycją trwają i powinny być zakończone przed końcem tego roku.

Cześć uwag dotyczyło wielkości czcionki. W pierwszym wydaniu użyłem rozmiaru czcionki 10 punktów, dzięki czemu książka była mniejsza i tym samym tańsza. Ponieważ jednak wiele osób narzekało na niewielkie literki, dlatego w tym wydaniu użyłem czcionki o rozmiarze 11 punktów co zwiększyło liczbę stron, ale poprawiło czytelność.

Poza rozmiarem czcionki wprowadziłem następujące zmiany:

- rozbudowałem rozdział poświęcony pisaniu wydajnych skryptów w programie R, rosnące rozmiary zbiorów danych wymuszają liczenie się z czasem obliczeń,
- dodałem rozdział poświęcony tworzeniu własnych pakietów,
- rozbudowałem rozdział poświęcony generowaniu zmiennych losowych,
- dodałem dwa podrozdziały opisujące funkcje graficzne i filozofię tworzenia wykresów w pakietach `lattice` i `ggplot2`,
- sprawdziłem czy w wersji R 2.15.1 (aktualnie najnowsza) działają wszystkie opisane w tej książce funkcje.

Usunąłem też różne drobne usterki, które zostały mi wskazane przez uważnych czytelników. Pewnie też wprowadziłem sporo nowych usterek, za wyśledzenie których będę wdzięczny uważnym czytelnikom.

Na koniec chciałbym szczególnie podziękować kilku osobom, które pośrednio lub bezpośrednio bardzo mi pomogły w pracy nad drugim wydaniem „Przewodnika ...". W pierwszej kolejności dziękuję żonie, Karolinie Biecek, bez której nieustannego wsparcia nie dałbym rady ani przygotować drugiego wydania ani zrealizować wielu innych projektów. Za wiele propozycji usprawnień, modyfikacji, korekt i rozszerzeń pierwszego wydania chciałbym podziękować przyjacielowi i współpracownikowi dr hab. Pawłowi Mackiewiczowi. Bardzo serdecznie chciałbym podziękować dr. Maciejowi Michalewiczowi i Justinowi Lindsleyowi, szefom nzLabs, oddziału badawczego firmy Netezza, za pomoc w finansowaniu konferencji WZUR (Warszawski/Wrocławski Zlot Użytkowników R), pomoc w finansowaniu mojego udziału w konferencjach use!R i umożliwienie mi realizacji ciekawych projektów rozwojowych dotyczących programu R. Dzięki tej współpracy łatwiej mi

zrozumieć jakie są oczekiwania dużych firm informatycznych co do modułów analitycznych oraz jak w tych zastosowaniach odnajduje się program R. Gorąco chciałbym podziękować również najlepszym wydawcom pod słońcem, panom dr. Marianowi Gewertowi i doc. dr. Zbigniewowi Skoczylasowi, za pomoc przy edycji i korekcie tak pierwszego jak i drugiego wydania oraz olbrzymi kapitał zaufania bez którego ta książka nigdy by się nie ukazała.

*Przemysław Biecek, Warszawa 2011*

## Przedmowa do wydania pierwszego

Szanowny Czytelniku, trzymasz właśnie w ręku książkę od początku do końca poświęconą pakietowi R. Książka ta powstała po to, by zaprezentować szeroki wachlarz możliwości pakietu R i ułatwić poznanie jego prostych i zaawansowanych aspektów. W sposób systematyczny przedstawia język R, na licznych przykładach opisuje podstawowe funkcje, prezentuje przydatne biblioteki dostępne w tym środowisku, opisuje popularne procedury statystyczne oraz funkcje do tworzenia grafiki.

Pozycja ta zaczęła powstawać w roku 2006, jako materiały pomocnicze dla moich studentów dzielnie poznających tajniki statystyki i analizy danych. Została rozbudowana i uzupełniona, aby mogła z niej skorzystać szersza grupa odbiorców. Starałem się wybrać materiał tak, by tę książkę chciały przeczytać:

- osoby, które chcą poznać pakiet R od podstaw, słyszały że warto i szukają łagodnego wprowadzenia dla zupełnych laików,
- osoby korzystające już z R, znające podstawy i chcące swoją wiedzę usystematyzować, uzupełnić, rozszerzyć, pogłębić,
- osoby pracujące z R na co dzień (eksperci), szukające podręcznej ściągawki (trudno spamiętać np. nazwy wszystkich argumentów graficznych) lub też chcące upewnić się, że o R wiedzą już (prawie) wszystko.

Innymi słowy, mam nadzieję, że każdy znajdzie tu coś dla siebie.

Książka podzielona jest na cztery części. Pierwsza część, to skrótowe przedstawienie możliwości pakietu R. Rozpoczyna się od wprowadzenia dla zupełnych nowicjuszy, ale w miarę upływu stron przedstawiane są kolejne, coraz bardziej zaawansowane informacje o języku oraz pakiecie R. Ta część jest przygotowana z myślą o osobach początkujących i o osobach chcących swoją wiedzę o R uzupełnić. Nie jest zakładana jakakolwiek wstępna wiedza o pakiecie R. Zaczynamy od podstaw, ale jestem pewien, że również spore grono zaawansowanych użytkowników znajdzie tutaj coś nowego. Dlatego warto przejrzeć tę część bez względu na stopień zaawansowania.

Kolejne części mają charakter encyklopedyczny i można je czytać w dowolnej kolejności. Część druga „pazuRrry" przedstawia możliwości języka R, o których warto wiedzieć i z których warto korzystać, a które nie znalazły się w innych częściach.

Najsilniejszą stroną programu R jest potężne wsparcie dla szeroko pojętych analiz statystycznych. W części trzeciej pt. „Wybrane procedury statystyczne" przedstawiono listę funkcji statystycznych wykorzystywanych przy najpopularniejszych procedurach statystycznych wraz z informacją, jak z tych funkcji korzystać i jak interpretować ich wyniki. Pakiet R świetnie nadaje się do tworzenia dobrze wyglądających rysunków, dlatego część czwarta „gRrrafika" poświęcona jest mechanizmom R umożliwiającym tworzenie i modyfikacje dobrze wyglądających wykresów (zarówno prostych jak i bardzo wymyślnych), schematów, grafik itp. Część czwarta kończy się prezentacją funkcji i argumentów graficznych, dzięki którym użytkownik ma pełną kontrolę nad tym co, jak i gdzie jest rysowane.

Program R rozwija się dynamicznie i nieustannie. Ma tak wiele możliwości, że nie sposób wszystkich opisać. Dołożyłem wszelkich starań, by ta pozycja była zrozumiała dla początkujących użytkowników i ciekawa dla użytkowników zaawansowanych. Będę zobowiązany czytelnikom za wszelkie uwagi i komentarze, które pozwolą uczynić tę pozycję czytelniejszą lub ciekawszą, zarówno te dotyczące zawartości jak i te dotyczące formy. Pod adresem `http://www.biecek.pl/R/R.pdf` znajdują się (w postaci elektronicznej) pierwsze 64 strony tej książki. Jest to, mam nadzieję, wystarczający fragment, by przekonać czytelnika, że warto bliżej zapoznać się z programem R. Ten fragment książki może być drukowany i kopiowany na użytek własny. Mam nadzieję, że pomoże on wielu osobom w pierwszym kontakcie z R, a także zachęci do nabycia całej książki w postaci drukowanej.

Książka ta mogła powstać wyłącznie dzięki mniejszej lub większej pomocy bardzo wielu osób, którym serdecznie dziękuję. Szczególnie gorąco dziękuję żonie Karolinie za jej wsparcie, wyrozumiałość, wytrwałość przy wielokrotnym czytaniu kolejnych wersji i moc cennych uwag. Wiele cennych wskazówek, sugestii, propozycji i uwag do kolejnych wersji otrzymałem od prof. dra hab. Jana Mielniczuka, który poświęcił bardzo wiele czasu korygując moje liczne pomyłki, serdecznie mu za to dziękuję. Za cenne uwagi merytoryczne chciałbym podziękować dr Janowi Ćwikowi i dr hab. Pawłowi Mackiewiczowi a również Grzegorzowi Hermanowiczowi i moim studentom, którzy czasem dzielili się uwagami, wątpliwościami oraz pomysłami na zmiany. Za pomoc przy wydawaniu tej książki chcę podziękować prof. dr hab. Jackowi Koronackiemu oraz wydawcom: dr. Marianowi Gewertowi i doc. dr. Zbigniewowi Skoczylasowi, bez których pomocy i zaangażowania książka ta nie powstałaby w postaci papierowej. Korzystając z okazji dziękuję moim wieloletnim współpracownikom dr inż. Adamowi Zagdańskiemu i dr inż. Arturowi Suchwałce za „zarażenie" mnie pakietem R i za wiele wspólnie realizowanych projektów wykonanych w programie R i nie tylko. Specjalne podziękowania składam również moim przełożonym: prof. dr hab. Teresie Ledwinie i prof. dr hab. Stanisławowi Cebratowi za pozostawienie mi swobody w wyborze zadań do realizacji.

To tyle tytułem wstępu. Życzę owocnej pracy z programem R.

*Przemysław Biecek, Wrocław 2008*

# Rozdział 1

# Łagodne wprowadzenie do R

## 1.1 Jak korzystać z tej książki?

Aby ułatwić wyszukiwanie informacji, pewne fragmenty tekstu zostały wyróżnioną innym krojem pisma lub dodatkowymi symbolami. Fragmenty skryptów w języku R oraz wyniki ich wykonania będą przedstawiane w następujących ramkach.

Tym sposobem kultowy przykład z „Hello world" mamy za sobą.

```
# Komentarz: moj pierwszy program.
for (i in 1:10) {
        cat("Hello world !!!\n")
}
```

Jeżeli razem z kodem R będzie pokazywany również wynik wykonania tego kodu, wtedy linie z wynikiem będą oznaczone podwójnym komentarzem ##.

R is the lingua franca of statistical research. Work in all other languages should be discouraged.

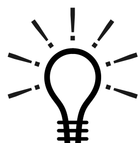
Jan de Leeuw
fortune(78)

```
substr("Co jest supeR?", start=13, stop=13)
## [1] "R"
```

Czasem tak bywa, że aż się prosi o komentarz do tekstu, nawet jeżeli nie jest to komentarz merytoryczny. Takie komentarze będą umieszczane na marginesie. Część z zamieszczonych na marginesie komentarzy to wybrane cytaty znanych użytkowników programu R. Jeżeli przypadną Wam one do gustu, to więcej cytatów znanych użytkowników programu R znaleźć można w pakiecie `fortunes`.

Fragmenty tekstu zasługujące na szczególną uwagę oraz komentarze do przedstawianego zagadnienia będą oznaczane krzywą opisaną równaniem w układzie biegunowym $G = \{(\rho, \phi) : \rho = 1 + 1/|\phi|, -\pi \leq \phi \leq \pi\}$ (czyli żarówką). Przykład poniżej.

Autor żyje w świecie liczb, wybaczcie mu brak poczucia humoru.
Przyp. żony.



Pamiętaj, żeby nie wychodzić z mokrą głową, gdy wieje silny wiatr!

Odnośniki do interesujących pozycji (zarówno w postaci papierowej jak i elektronicznej) zostały zgromadzone na końcu tej książki. Do pozycji książkowych lub adresów stron internetowych będziemy odnosili się podając pozycję źródła w spisie literatury. Podobnie jak w książce [1].

Pisząc o funkcjach, pakietach, elementach języka stosować będziemy `czcionkę o stałej szerokości znaków`. Podając angielskie nazwy będziemy je oznaczać *kursywą*. Przy pierwszym wymienieniu nazwy funkcji z niestandardowego pakietu zaznaczymy również w jakim pakiecie ta funkcja jest dostępna. Zapis `ggplot2::qplot()` oznacza, że funkcja `qplot()` znajduje się w pakiecie `ggplot2`. Czasem funkcja o tej samej nazwie dostępna jest w różnych pakietach, wskazując w ten sposób jawnie w którym pakiecie dana funkcja się znajduje pozwala na jednoznaczne określenie o którą funkcję chodzi. Również w sytuacji gdy do użycia danej funkcji potrzeba zainstalować lub włączyć niestandardowy pakiet, warto wiedzieć w jakim pakiecie danej funkcji szukać. Na końcu tej książki znajduje się indeks funkcji zarówno alfabetyczny jak i w podziale na pakiety.

Przy nauce nowych rzeczy bardzo przydatne są zadania, które można samodzielnie rozwiązać. Tak jest też w przypadku pakietu R, dlatego do każdego rozdziału przygotowana została lista zadań weryfikujących zdobytą wiedzę. Zadania umieszczone są na końcu każdego z rozdziałów i podzielone są ze względu na różne poziomy trudności. Pliki z przykładowymi odpowiedziami znajdują się pod adresem `http:\\www.biecek.pl\R\`. Na tej stronie umieszczane są również dodatkowe materiały ułatwiające poznawanie pakietu R.

## 1.2 Słów kilka o projekcie R

R to zarówno nazwa języka programowania, nazwa platformy programistycznej wyposażonej w interpreter tego języka oraz nazwa projektu, w ramach którego rozwijany jest zarówno język jak i środowisko. W dalszej części książki będziemy korzystali z nazwy R, mając na myśli tak język programowania, platformę programistyczną jak i zbiór bibliotek (pakietów), w które wyposażona jest ta platforma.

R jest często nazywany pakietem statystycznym. Jest tak z uwagi na olbrzymią liczbę dostępnych funkcji statystycznych. Możliwości R są jednak znacznie większe. W Internecie można znaleźć przykłady wykorzystania R do automatycznego generowania raportów, wysyłania maili, rysowania fraktali, czy renderowania trójwymiarowych animacji. W tej książce skupimy się wyłącznie na najpopularniejszych możliwościach R. Jednak czytelnik, który dobrze pozna przedstawione w tej książce podstawy, nie będzie miał problemów ze zrozumieniem zasad funkcjonowania innych pakietów.

Pierwsza wersja R została napisana przez Roberta Gentlemana i Rossa Ihake (znanych jako R&R:) pracujących na Wydziale Statystyki Uniwersytetu w Auckland. Pakiet R początkowo służył jako pomoc dydaktyczna do uczenia statystyki na tym uniwersytecie. Jednocześnie, ponieważ był to projekt otwarty, bardzo szybko zyskiwał na popularności. Od roku 1997 rozwojem R kierował zespół ponad dwudziestu osób nazywanych *core team*. W zespole tym znaleźli się eksperci z różnych dziedzin (statystyki, matematyki, metod numerycznych oraz szeroko pojętej informatyki) z całego świata. Liczba osób rozwijających R szybko rosła, a aktualnie rozwojem projektu kieruje fundacja „The R Foundation for Statistical Computing" licząca setki aktywnych uczestników. Ponadto w rozwój R mają wkład tysiące osób z całego świata publikujące własne biblioteki/pakiety najróżniejszych funkcji znaj-

dujących zastosowania w wielu dziedzinach. Liczba bibliotek dostępnych dla użytkowników R szybko rośnie, przekraczając w poprzednim roku liczbę 4000. A jest też znacząca liczba pakietów dostępnych w nieformalnym obiegu.

Język R był wzorowany na języku S, który został opracowany w laboratoriach Bell'a. Z tego też powodu język R jest bardzo podobny do języka S. Programy w S działają pod R lub można je prosto zmodyfikować tak, by działały. Wiele funkcji w R ma dodatkowe argumenty dodane po to, by zapewnić zgodność z S. Dzięki temu, że języki R i S są do siebie podobne możemy wykorzystywać liczne książki do pakietu S do nauki języka R jak i do poznania dostępnych funkcji statystycznych. Bardzo dobrą książką do nauki języka S jest książka Johna Chambersa [6] a do nauki funkcji statystycznych dostępnych w programie S polecam pozycję Briana Everitta [3]. Uzupełnieniem do licznych pozycji książkowych jest olbrzymia liczba stron internetowych oraz dokumentów elektronicznych szczegółowo przedstawiających rozmaite aspekty programu i języka R. Pod koniec roku 2007 ukazała się bardzo obszerna i godna polecenia książka Michaela Crawleya [4] przedstawiająca zarówno język R jak i wiele procedur statystycznych zaimplementowanych w programie R. Pojawiają się też i będą się pojawiały liczne książki poświęcone specjalistycznym zastosowaniom programu R, jak np. świetna pozycja przygotowana przez Paula Murrella poświęcona grafice [33], książka autorstwa Juliana Farawaya poświęcona modelom liniowym [22], czy kolejna pozycja Briana Everitta przedstawiająca podstawy statystyki [21]. Wydawca Springer ma w ofercie ponad 45 książek o programie R wydanych w serii 'Use R!', każda z tych pozycji traktuje o jednym wybranym zagadnieniu, takim jak analizy danych przestrzennych, wizualizacja danych, analizy danych socjologicznych, genetycznych itp.

> Overall, SAS is about 11 years behind R and S-Plus in statistical capabilities (last year it was about 10 years behind) in my estimation.
>
> Frank Harrell (SAS User, 1969-1991) fortune(10)

Przejście z języka S na język R jest bardzo proste. Również osoby korzystające z innych platform statystycznych takich jak Matlab, Octave, SPSS, SAS itp. nie będą miały większych problemów z przestawieniem się na pakiet R. Istnieje wiele dokumentów przedstawiających różnice pomiędzy danym językiem a R oraz zawierających rady dla użytkowników innych pakietów jak szybko zacząć korzystać z R. Listę wielu przydatnych wskazówek znajdziemy pod adresem [2].

Program R jest projektem GNU opartym o licencje GNU GPL, oznacza to, iż jest w zupełności darmowy zarówno do zastosowań edukacyjnych jak i biznesowych. Więcej o licencji GNU GPL można przeczytać pod adresem [5]. Platforma R wyposażona jest w świetną dokumentację, dostępną w postaci dokumentów pdf lub stron html. Aktualnie dokumentacja ta w większości jest angielskojęzyczna, niektóre pliki pomocy mają już swoje lokalizacje, w tym polską.

> Programy napisane w językach takich jak C, C++, Pascal itp. można kompilować. Programy skompilowane do rozkazów rozumianych bezpośrednio przez procesor są z reguły szybsze, ale programy z reguły trudniej napisać i trwa to dłużej. Języki interpretowane (skryptowe) nadają się świetnie do szybkiego pisania programów, w sytuacji, gdy czas wykonania nie jest kluczowy.

Język R jest językiem interpretowanym a nie kompilowanym. Korzystanie z R sprowadza się do podania ciągu komend, które mają zostać wykonane. Kolejne komendy mogą być wprowadzane linia po linii lub też mogą być wykonywane jako skrypt (czyli plik tekstowy z zapisaną listą komend do wykonania). Wiele osób uważa (często słusznie), że języki interpretowane są wolne i wymagają dużo pamięci, jednak obecne możliwości komputerów pozwalają w standardowych zastosowaniach zupełnie się tym nie przejmować. Ponadto istnieje wiele dodatkowych narzędzi pozwalających na kompilowanie kodu R np. podczas wykonywania (ang. *just in time compilation*), rozwiązania zwiększające wydajność programu R przedstawione są w rozdziale 2.9.3.1.

GUI to skrót od ang. *Graphical User Interface*, czyli graficznego interfejsu użytkownika.

Osoby, które nie chcą pamiętać składni komend R mogą skorzystać z istniejących nakładek i GUI. Przykładowo, korzystając z okienkowego interfejsu pakietu `Rcmdr` można wyklikać wiele różnych raportów statystycznych, podsumowań i wykresów. Pakietów wspierających takie klikane analizy jest znacznie więcej, np. `Deducer`, `rattle` itp. Zdecydowanie jednak zachęcam do przełamania niechęci do pamiętania i wpisywania komend. Naprawdę warto samodzielnie przygotowywać i modyfikować skrypty! Po pewnym czasie staje się to proste i umożliwia dużą automatyzację pracy oraz znaczne zaoszczędzenie czasu.

Pierwszy podrozdział zakończę przedstawieniem czterech głównych (ale nie jedynych) zalet platformy R. W skrócie, są to cztery powody dla których program R deklasuje konkurencję.

- Program R pozwala na tworzenie i upowszechnianie pakietów zawierających nowe funkcjonalności. Obecnie dostępnych jest przeszło 4000 pakietów do różnorodnych zastosowań, np. `rgl` do grafiki trójwymiarowej, `lima` do analizy danych mikromacierzowych, `seqinr` do analizy danych genomicznych, `psy` z funkcjami statystycznymi popularnie wykorzystywanymi w psychometrii, `geoR` z funkcjami geostatystycznymi, `Sweave` do generowania raportów w języku LaTeX i wiele, wiele innych. Każdy możne napisać swój własny pakiet i udostępnić go innym osobom.
- Program R pozwala na wykonywanie funkcji z bibliotek napisanych w innych językach (C, C++, Fortran) oraz na wykonywanie funkcji dostępnych w programie R z poziomu innych języków (Java, C, C++, pakiety Statistica, SAS i wiele innych). Dzięki temu możemy np. znaczną część programu napisać w Javie, a R wykorzystywać jako dużą zewnętrzną bibliotekę funkcji statystycznych.
- Program R jest w zupełności darmowy do wszelkich zastosowań zarówno prywatnych, naukowych jak i komercyjnych. Również większość pakietów napisanych dla R jest darmowych i dostępnych w ramach licencji GNU GPL lub GNU GPL 2.0.
- W programie R można wykonać wykresy o wysokiej jakości, co jest bardzo istotne przy prezentacji wyników. Wykresy te nawet przy domyślnych ustawieniach wyglądają znacznie lepiej od podobnych wykresów przygotowanych w innych pakietach.

Panie, takie rzeczy to tylko w eRze

## 1.3 Instalacja

Instalacja pakietu R składa się z dwóch etapów. Pierwszy, to zainstalowanie podstawowego środowiska (tzw. *base version*) wraz z podstawowymi bibliotekami. Ten podstawowy zestaw już ma potężne możliwości w większości przypadków wystarczające do analizy danych, rysowania wykresów i wykonywania innych typowych zadań. Drugi etap, to uzupełnianie wersji podstawowej przez doinstalowanie pakietów z przydatnymi funkcjami. Aktualnie dostępnych jest ponad cztery tysiące pakietów! Nie ma jednak potrzeby instalowania ich wszystkich od razu. Z regu-

ły w miarę używania okazuje się, że przydałaby się nam jakaś dodatkowa funkcja, która jest już dostępna w pewnym pakiecie i dopiero wtedy warto taki pakiet doinstalować. Poniżej znajduje się krótka informacja jak łatwo przebrnąć przez oba etapy instalacji.

## Instalacja środowiska

Dla większości systemów operacyjnych, w tym wszystkich dystrybucji Linuxa, Unixa, dla wersji Windowsa począwszy od Windowsa 95 i dla MacOSa, pakiet R jest dostępny w postaci źródłowej oraz skompilowanej. Najłatwiej jest zainstalować R korzystając ze skompilowanego pliku instalacyjnego. Instalacja jest prosta, wystarczy wybrać jeden z serwerów mirror, na którym umieszczony jest plik instalacyjny, ściągnąć ten plik, uruchomić go, a następnie postępować zgodnie z instrukcjami. Adresy serwerów z kopiami plików instalacyjnych pakietu R znaleźć można pod adresem `http://cran.r-project.org/mirrors.html`. W większości przypadków najszybciej ściągniemy program R z wrocławskiego serwera [7].

Mirror to tzw. serwer lustrzany, w którym znajduje się dokładna (lustrzana) kopia plików. Jeżeli chcemy ściągnąć pliki z serwera, który jest daleko od naszego komputera i z którego korzysta wiele osób to ściąganie będzie wolne. Dlatego warto wybrać serwer położony możliwie blisko, o małym obciążeniu.

Szczegółową instrukcję instalacji można znaleźć pod adresem [8], przyda się ona osobom chcącym zainstalować nietypową konfigurację R. W dalszej części będzie opisywana 64 bitowa wersja pakietu R przygotowana dla systemu Windows. Na dzień dzisiejszy najnowszą wersją jest 3.0.1. Aby przystąpić do instalacji należy uruchomić plik `R-3.0.1-win.exe`. Cała instalacja ogranicza się praktycznie do klikania przycisku „Next". Po zainstalowaniu R utworzy we wskazanym miejscu (najczęściej będzie to katalog `c:/Program Files/R/R-3.0.1`) strukturę podkatalogów z plikami potrzebnymi do działania.

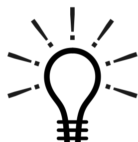
Po instalacji w utworzonej strukturze znajdą się różne podkatalogi. W tym: katalog `bin` (z plikami wykonywalnymi R), `doc` (z ogólną dokumentacją R), `library` (w którym instalowane są kolejne pakiety) i innymi, mniej ważnymi. Platformę R można uruchomić w trybie tekstowym (uruchamiając plik `R.exe`) lub też w trybie z prostym okienkowym GUI (uruchamiając plik `Rgui.exe`). Oba pliki do uruchomienia środowiska znajdują się w katalogu `bin`. Wersja tekstowa może się przydać, jeżeli nie chcemy tracić zasobów na inicjowanie interfejsu graficznego. Wybór trybu uruchomienia proponujemy oprzeć na prostej zasadzie: jeżeli nie wiesz, czym te tryby się różnią, to uruchom `Rgui.exe`.

Począwszy od wersji 2.12.0 plik Rgui.exe znajduje się w katalogu bin/i386 dla wersji 32 bitowej i bin/x64 dla wersji 64 bitowej.

Wygodną właściwością środowiska R jest to, że można je uruchamiać bez instalowania. Można więc skopiować środowisko R na płytę CD, na pendrive lub dysk przenośny i uruchamiać na dowolnym komputerze bez potrzeby instalacji.

Trudno jest podać minimalne wymagania sprzętowe niezbędne do działania R. Jeszcze nie zdarzyło mi się nie móc uruchomić tego pakietu na napotkanym komputerze. Można śmiało przyjąć, że 256MB RAM, procesor klasy Pentium lub wyższej i kilkadziesiąt MB miejsca na dysku twardym w zupełności wystarczą. Do pełnego komfortu przyda się szybszy procesor, więcej RAM i więcej miejsca na dysku twardym (bioinformatyczne zbiory danych potrafią zajmować bardzo dużo miejsca na dysku i w RAM). Program R przechowuje dane w pamięci RAM, więc do analiz dużych zbiorów danych potrzebne jest dużo wolnej pamięci RAM. Jeżeli

dysponujemy maszyna wyposażoną w więcej niż 2GB RAM to powinniśmy używać 64bitowej wersji R. Pierwsza oficjalna wersja 64bitowa to 2.11.0. Wersja 64 bitowa umożliwia użycie całej dostępnej pamięci operacyjnej, w przypadku wersji 32bitowej wykorzystać można jedynie od 2 do 4GB w zależności od systemu operacyjnego



Osoby używające programu R do bardzo wymagających obliczeniowo analiz powinny raczej używać Linuxowej lub Unixowej wersji R. W tych systemach operacyjnych zarządzanie pamięcią jest wydajniejsze przez co R działa (odrobinę) szybciej. Dla linuxów dostępne są też dodatkowe narzędzia pozwalające na wykorzystanie wielowątkowości i innych mechanizmów systemowych (np. funkcji `fork()`).

## Instalacja i ładowanie pakietów

Jak już pisaliśmy, po zainstalowaniu podstawowego zbioru bibliotek program R ma już spore możliwości. Prawdziwa potęga kryje się w setkach dodatkowych pakietów, w których znajdują się tysiące różnych funkcji (funkcje w programie R pogrupowane są w pakietach/bibliotekach). Po uruchomieniu systemu R kolejne pakiety można zainstalować funkcją `install.packages()`. Poniższe polecenie instaluje pakiet `Rcmdr` wraz z pakietami zależnymi, wymaganymi do jego działania.

```
install.packages("Rcmdr", dependencies = TRUE)
```

Można też zainstalować nowy pakiet wybierając z menu opcję `install package(s)`. Przy instalacji pierwszego pakietu zostaniemy zapytani z jakiego serwera lustrzanego chcemy korzystać.

Po zainstalowaniu nowego pakietu, pliki z danymi, funkcjami i plikami pomocy znajdą się na dysku twardym komputera. Wszystkie pakiety są wgrywane jako podkatalogi do katalogu `library`. Aby móc skorzystać z wybranych funkcji należy przed pierwszym użyciem załadować (włączyć) odpowiedni pakiet. Po każdym uruchomieniu platformy R ładowane są pakiety podstawowe takie jak: `base`, `graphics`, `stats`, itp. Aby skorzystać z dodatkowych funkcji lub zbiorów danych, należy załadować (włączyć) pakiet, w którym się one znajdują (zakładamy, że pakiety te zostały już zainstalowane). Pakiety włącza się poleceniem `library()`.

Poniższa instrukcja włącza pakiet `Rcmdr`. Gdyby ten pakiet nie był zainstalowany, to pojawiłby się poniżej przedstawiony komentarz.

```
library(Rcmdr)
## Error in library(Rcmdr) : there is no package called 'Rcmdr'
```

Jak już pisaliśmy, aktualnie dostępnych jest ponad 3500 pakietów, które możemy dodatkowo zainstalować. W tak dużym zbiorze trudno czasem odnaleźć pakiet z interesującą nas funkcjonalnością. Dlatego, przedstawiając nowe funkcje będziemy korzystać z notacji `nazwaPakietu::nazwaFunkcji()`. Zapis `ggplot2::qplot()` oznacza, że funkcja `qplot()` znajduje się w pakiecie `ggplot2`. Ten prefix pomijamy dla pakietów `base`, `stats`, `utils` i `graphics`, których nie trzeba dodatkowo włączać, są automatycznie włączane z chwilą startu środowiska R. W skorowidzu, znajdującym się na końcu książki, funkcje są wymienione zarówno w kolejności

alfabetycznej jak i po pogrupowaniu w pakiety. Jeżeli znamy nazwę funkcji i chcemy dowiedzieć się w jakim pakiecie ta funkcja się znajduje a nie mamy tej książki pod ręką, to możemy wykorzystać funkcją `help.search()`. Przeszuka ona wszystkie zainstalowane pakiety w poszukiwaniu funkcji o wskazanej nazwie lub funkcji, w których opisie wystąpiło zadane słowo kluczowe. Więcej o tej funkcji i innych sposobach wyszukiwania informacji o funkcjach napiszemy w podrozdziale 1.5.2.

Po załadowaniu odpowiedniego pakietu możemy korzystać z dostępnych w nim funkcji podając ich nazwę. Możemy też ręcznie wskazać, z którego pakietu funkcję chcemy uruchomić, co jest przydatne, gdy funkcje o identycznych nazwach znajdują się w kilku załadowanych pakietach. Przykładowo zarówno w pakiecie `epitools` jak i `vcd` znajduje się funkcja `oddsratio()` o tej samej nazwie ale innym działaniu. Aby wskazać, z którego pakietu chcemy wybrać funkcję należy użyć operatora :: lub :::. Operator :: pozwala na odnoszenie się wyłącznie do publicznych funkcji z pakietu, podczas gdy operator ::: umożliwia odnoszenie się również do funkcji prywatnych.

Oba poniższe wywołania dotyczą funkcji `seq()` z pakietu `base`, drugi sposób jest szczególnie przydatny, gdy występuje kolizja nazw funkcji z różnych pakietów.

```
seq(10)
base::seq(10)
```

Jeżeli nie użyjemy tego operatora, a dojdzie do kolizji nazw, to program R użyje funkcji z ostatnio załadowanego pakietu.

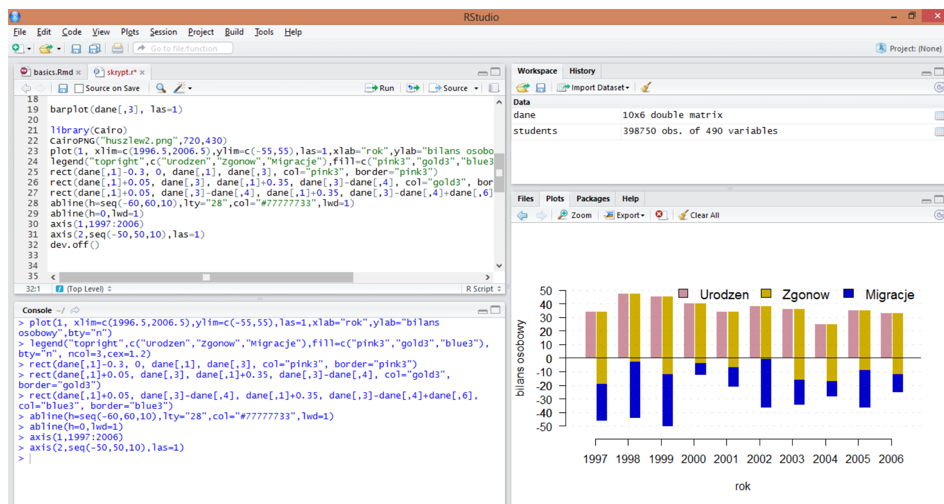
## 1.4 RStudio, uniwersalny edytoR

Jeżeli wykorzystujemy R do prostych obliczeń, nie piszemy własnych funkcji i nie zależy nam na powtarzaniu wykonywanych analiz, to możemy komendy wpisywać bezpośrednio w linii komend konsoli R. Jednak przy większych programach lub gdy zależy nam na możliwości powtarzania analiz, potrzebny nam będzie edytor, w którym będziemy mogli tworzyć i edytować skrypty R. Listę ponad dwudziestu różnych edytorów wspierających składnię języka R przedstawiono na stronie [41]. W ostatnich latach największą i wciąż rosnącą popularność zdobył bezpłatny edytor RStudio opracowany przez firmę o tej samej nazwie. Ten edytor dostępny jest zarówno na platformę Windows, Linux jak i MacOS, wspiera on wiele zaawansowanych pakietów i funkcjonalności R. Obecnie deklasuje on konkurencje, nie będziemy więc pisać o konkurencji. RStudio pobrać ze strony `http://www.rstudio.com/`. Przykładowy wygląd tego programu jest przedstawiony na rysunku 1.1).

Po zainstalowaniu RStudio domyślnie wykorzystuje najnowszą z zainstalowanych wersji programu R. Można to oczywiście zmienić w ustawieniach.

Korzystanie z tego edytora jest bardzo intuicyjne. Z ciekawszych funkcji można wymienić:

- zarządzanie wieloma plikami/projektami,
- możliwość automatycznego wysyłana całego skryptu lub fragmentu kodu do konsoli R. Po zaznaczeniu określonego fragmentu kodu po naciśnięciu `Ctrl-Enter` zadany fragment będzie wykonany w konsoli R.
- wyświetlanie obiektów (nazwa, klasa, wymiary) obecnych w przestrzeni nazw,




**Rysunek 1.1:** Przykładowy wygląd paneli edytora `RStudio`. Na czterech panelach widoczne są: okno pozwalające na edycje kodu, okno pozwalające na podgląd zmiennych załadowanych do globalnej przestrzeni nazw R, konsola programu R do której możemy wpisywać bezpośrednio polecenia oraz okno w którym domyślnie wyświetlane są pliki pomocy oraz wykresy.

- edytor danych, funkcji i innych obiektów R, po kliknięciu na nazwę zbioru danych obecnego w przestrzeni nazw mamy możliwość edycji tego obiektu, jest to często wygodniejszy sposób niż użycie funkcji `fix()` czy `edit()`,
- uproszczony sposób wczytywania danych z plików poprzez menu,
- podświetlanie słów kluczowych i funkcji,
- kontekstowe uzupełnianie nazw funkcji, zmiennych, właściwości, argumentów funkcji (gdy rozpoczynamy pisanie nazwy funkcji, zmiennej lub argumenty pojawia się lista wszystkich słów, które pasują w danym kontekście),
- zwijanie/rozwijanie kodu funkcji i pętli,
- wsparcie dla pakietów `knitra`, `Sweave` i `shiny`, łatwa nawigacja po wstawkach kodu,
- domykanie otwartych nawiasów, cudzysłowów, wraz z inteligentnym zaznaczaniem zawartości (dwukrotne kliknięcie we wnętrze nawiasu, zaznacza cała zawartość nawiasu),
- inteligentne wstawianie wcięć połączone z rozpoznawaniem składni (czyli nowe wcięcie dodawane jest w pętlach, funkcjach itp),
- interaktywny debugger (na dzień dzisiejszy dostępny jedynie w wersji deweloperskiej, ale niedługo będzie też dostępny w wersji oficjalnej).

## 1.5 Startujemy

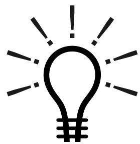
Zakładamy, że czytelnik ma już zainstalowany na dysku program R. Warto na bieżąco i własnoręcznie sprawdzać na komputerze reakcje R na opisywane w tej książce polecenia. Jeżeli jakiś fragment nie jest zrozumiały, proszę pominąć go i czytać dalej. Niektóre komentarze i uwagi przeznaczone są dla odrobinę bardziej zaawansowanych czytelników, nie ma się więc co zrażać, jeżeli nie wszystko będzie jasne przy pierwszym czytaniu.

### 1.5.1 Pierwsze uruchomienie

Po zainstalowaniu programu R, czas na pierwsze jego uruchomienie. W systemie Windows najlepiej uruchomić plik `Rgui.exe` z katalogu `bin` lub `bin\x64` dla R w wersji 3.0.x. Uruchamia on R z wbudowanym interfejsem graficznym. Program R można uruchomić również w trybie wsadowym lub trybie tekstowym, ale to jest temat, który omówimy w rozdziale 2.5. Polecenie `Rgui` nie działa pod systemem Linux, w tym przypadku R możemy uruchomić poleceniem `R` lub korzystając z innego interfejsu graficznego. W tym i kolejnym podrozdziale będą przedstawiane przykłady działania programu `Rgui.exe` w wersji dla Windows. Nazwy funkcji i argumentów nie zależą od systemu operacyjnego.

Po uruchomieniu R pojawi się ekran powitalny oraz wyświetli się znak zachęty >. Znak ten oznacza, że platforma R jest gotowa do realizacji kolejnego polecenia. Efekt uruchomienia okienkowej wersji R przedstawiony jest na Rysunku 1.2.

Tak jak pisaliśmy w rozdziale *edytoR*, przyjemniej się pracuje z programem RStudio niż z „surowym” R.



Znak > jest znakiem zachęty do wprowadzenia kolejnych poleceń. Jest wyświetlany tylko, gdy platforma zakończyła już wykonywanie polecenia wprowadzonego w poprzedniej linii. Jeżeli nowa linia rozpoczyna się znakiem + (znakiem kontynuacji), to znaczy, że polecenie wpisane w poprzedniej linii nie zostało jeszcze zakończone i platforma czeka na dalszą jego część (np. rozpoczęta jest pętla, otwarty jest nawias lub cudzysłów). Jeżeli nowa linia nie rozpoczyna się żadnym znakiem, to znaczy, że R jest w trakcie wykonywania jakiegoś czasochłonnego polecenia lub też czeka na reakcje użytkownika (kliknięcie myszką lub naciśnięcie któregoś klawisza na klawiaturze). Jeżeli nie wiemy na co R czeka, to klawiszem `ESC` przerywamy aktualnie wykonywaną przez R czynność i wracamy do znaku zachęty.

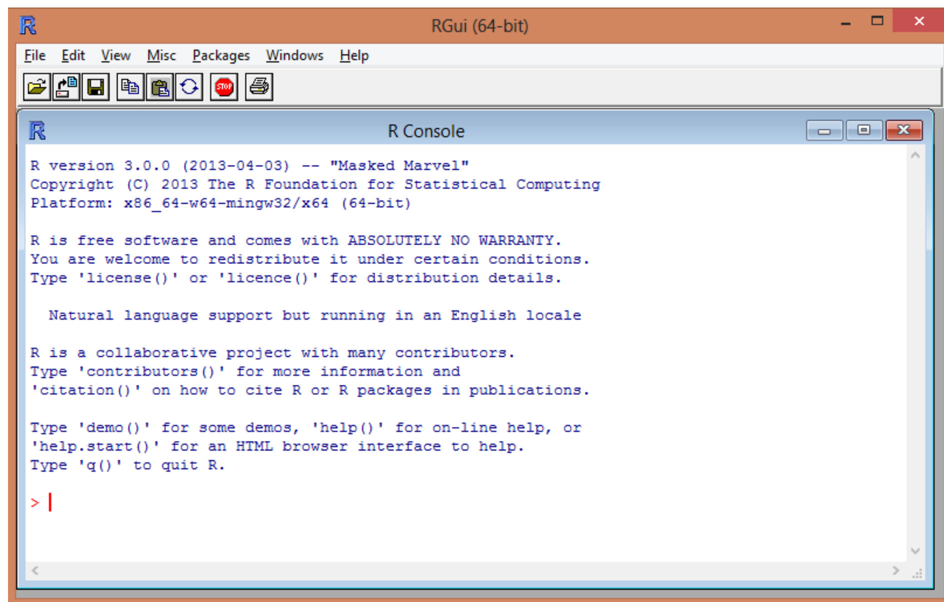
Pierwsze polecenie, które warto przećwiczyć to zamknięcie platformy R.

```
q()
```

Po wykonaniu tego polecenia zostaniemy zapytani, czy zachować aktualny stan pracy, następnie środowisko R zostanie zamknięte.

Jeszcze nic nie zrobiliśmy, więc można nie zachowywać stanu pracy.

Potrafimy już zamknąć program R, spróbujmy teraz znaleźć motywację do dalszej nauki. Dla wielu pakietów oraz funkcji dostępnych w programie R zostały przygotowane prezentacje, pokazujące możliwości danego pakietu lub funkcji. Takie prezentacje uruchamia się funkcją `demo(utils)`. Zobaczmy kilka ciekawszych prezentacji! Aby to zrobić należy wpisać do konsoli jedną z następujących linii a następnie nacisnąć klawisz `ENTER`.




**Rysunek 1.2:** Okno powitalne, otrzymane po uruchomieniu programu R.

Poniższym poleceniem uruchamiamy „klikany" interfejs R.

```
library(Rcmdr)
```

A teraz uruchomimy kilka prezentacji dla funkcji `persp()` (rysowanie rzutów), pakietu `graphics`, znaków Kanji.

```
demo(persp)
demo(graphics)
demo(Japanese)
```

Aby zobaczyć prezentację pakietu `rgl` (grafika trójwymiarowa) musimy najpierw ten pakiet włączyć.

```
library(rgl)
demo(rgl)
```

Teraz powinniśmy być już wystarczająco zmotywowani. Kolejny podrozdział przedstawia poszczególne opcje menu w okienkowej wersji R.

R will always be arcane to those who do not make a serious effort to learn it. It is **not** meant to be intuitive and easy for casual users to just plunge into. It is far too complex and powerful for that. But the rewards are great for serious data analysts who put in the effort.

Berton Gunter
fortune(196)

## 1.5.2 Gdzie szukać pomocy?

Gdy nie wiemy jak coś zrobić, to najłatwiej i najszybciej jest zapytać się kogoś, kto to wie i chce nam podpowiedzieć. W sytuacji, gdy nie mamy takiej osoby pod ręką R oferuje bogaty system pomocy.

Pierwszym źródłem pomocy są wbudowane funkcje R ułatwiające wyszukiwanie informacji. Oto lista najbardziej przydatnych:

- Funkcja `help()` wyświetla stronę powitalną systemu pomocy R. Na tej stronie opisane są szczegółowo wymienione poniżej funkcje.

- Funkcje `help("nazwaFunkcji")` i `?nazwaFunkcji` wyświetlają stronę z pomocą dla funkcji o nazwie `nazwaFunkcji`. Zobacz też przykład przedstawiony poniżej. Format opisów funkcji jest ujednolicony, tak aby łatwo było z nich korzystać. Kolejne sekcje pomocy zawierają: zwięzły opis funkcji (sekcja `Description`), deklaracje funkcji (sekcja `Usage`), objaśnienie poszczególnych argumentów (sekcja `Arguments`), szczegółowy opis funkcji (sekcja `Details`), literaturę (sekcja `References`), odnośniki do innych funkcji (sekcja `See Also`) oraz przykłady użycia (sekcja `Examples`). Jeżeli podamy argument `package`, to uzyskamy pomoc dotyczącą wskazanego pakietu. Przykładowo aby wyświetlić opis pakietu `MASS` należy użyć polecenia `help(package=MASS)`.
- Funkcja `args("nazwaFunkcji")` wyświetla listę argumentów `nazwaFunkcji`.
- Funkcje `apropos(slowo)` i `find(slowo)` wypisują listę funkcji (oraz obiektów), które w swojej nazwie mają podciąg `slowo`.
- Funkcja `example("nazwaFunkcji")` uruchamia skrypt z przykładami dla funkcji `nazwaFunkcji`. Dzięki przykładom można szybko zobaczyć jak korzystać z danej funkcji, a także jakich wyników się należy spodziewać. Na dobry początek warto sprawdzić wynik polecenia `example(plot)`.
- Funkcja `help.search("slowoKluczowe")` przegląda opisy funkcji znajdujących się w zainstalowanych pakietach i wyświetla te funkcje, w których znaleziono wskazane `slowoKluczowe`. W tym przypadku `slowoKluczowe` może oznaczać również kilka słów lub zwrot. W liście wyników znajduje się również informacja, w którym pakiecie znajdują się znalezione funkcje.

Poniżej przedstawiona jest przykładowa sesja w programie R. Poszukujemy informacji o funkcji `plot()` oraz o funkcjach do testowania hipotez. W pierwszej linii wyświetlamy pomoc dotyczącą funkcji plot(), następnie przykłady użycia funkcji plot(). Kolejna linia wyświetla funkcje ze słowem `"test"` w nazwie a ostania wyświetla nazwy funkcji ze zwrotem `"normality test"` w opisie.

```
?plot
example(plot)
apropos("test")
help.search("normality test")
```

Funkcje przedstawione powyżej wyszukują informacje na zadany temat wśród pakietów, które są już zainstalowane na komputerze. Jeżeli to okaże się niewystarczające (a może się zdarzyć, że nie mamy zainstalowanego pakietu, w którym znajduje się potencjalnie interesująca nas funkcja), to możemy skorzystać z zasobów dostępnych w Internecie. W szczególności warto wiedzieć gdzie znaleźć:

Szukając informacji związanych z R w wyszukiwarkach, warto do wyszukiwanego hasła poza R dodać również napis CRAN (The Comprehensive R Archive Network). Ułatwi to znalezienie informacji związanych z „tym” znaczeniem literki R.

- Poradniki (podręczniki, ang. *manuals*), poświęcone różnym aspektom programowania w programie R lub analizy danych w programie R. Dostępne są bezpośrednio z menu `Help` w programie R oraz w Internecie pod adresem `http://cran.r-project.org/manuals.html`.

- R-bloggers, czyli agregator blogów poświęconych programowi R, często źródło wielu ciekawych informacji `http://www.r-bloggers.com`.
- Książki poświęcone pakietowi R oraz o analizie danych z użyciem tego pakietu. Aktualizowana lista książek na ten temat znajduje się online pod adresem `http://www.r-project.org/doc/bib/R-books.html`.
- Portale z pytaniami i odpowiedziami dotyczącymi statystyki, programowania. Np. `http://stats.stackexchange.com` (pytania dotyczące statystyki) lub `http://stackoverflow.com/questions/tagged/r` (pytania dotyczące programowania).
- FAQ dostępne pod adresem `http://cran.r-project.org/faqs.html`. Tysiące lub setki tysięcy osób używa R, więc pewne pytania zostały już zadane setki razy. FAQ, to miejsce, w którym znajdziesz odpowiedzi na najczęstsze pytania (stąd też nazwa FAQ, skrót od ang. *Frequently Asked Question*).

Powyższe źródła są bez wyjątku angielskojęzyczne. Poza nimi w Internecie można znaleźć też wiele materiałów polskojęzycznych. Jednym z pierwszych jest *„Wprowadzenie do środowiska R”* Łukasza Komsty [13]. Ostatnio pojawia się coraz więcej polskojęzycznych książek dotyczących wybranych zastosowań pakietu R. W razie napotkania problemów można zadać pytanie na którymś z polskich forów, na którym wypowiadają się użytkownicy programu R, np. na googlowej „Polskiej grupie użytkowników programu R” [9].

### 1.5.3 kalkuRator

Program R to bardzo potężny, zaawansowany i rozbudowany pakiet statystyczny. Ale można też korzystać z niego tak, jak z bardzo rozbudowanego kalkulatora. Zacznijmy od kilku prostych działań. Poniższa ramka przedstawia wynik przykładowej sesji z R. Po znaku zachęty `">"` znajdują się wprowadzone polecenia. Naciśnięcie klawisza `ENTER` powoduje zakończenie linii i (o ile to możliwe) wykonanie polecenia.

Poniżej przedstawiamy przykładową sesję z pakietem R w roli kalkulatora. Zacznijmy od kilku prostych operacji arytmetycznych.

```
2+2
## [1] 4
2^10 - 1
## [1] 1023
1 / 5
## [1] 0.2
(3 + 7)^(4 - 2)
## [1] 100
log(1024, 2)
## [1] 10
```

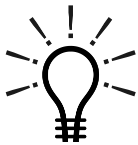
Funkcje trygonometryczne operują na radianach.

```
sin(pi/2)
## [1] 1
```

```
sin(pi/3)^2 + cos(pi/3)^2
## [1] 1
atan2(1,1)
## [1] 0.7853982
```

Coś trudniejszego, symbol Newtona, tylko w dobrych kalkulatorach.

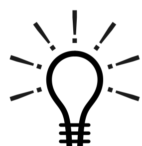
```
choose(6,2)
## [1] 15
```



Napis `[1]` rozpoczynający linię z wynikiem związany jest ze sposobem działania funkcji `print()` wyświetlającej obiekty, w tym przypadku wektory liczb. Mianowicie, jeżeli wyświetlane są wartości wektora, to w nawiasie kwadratowym znajduje się indeks elementu wyświetlanego bezpośrednio za tym nawiasem. Dzięki temu jeżeli wartości wektora rozpisane są w wielu liniach, to dla każdej linii wiadomo który element jest pierwszym wyświetlanym. W prezentowanych przypadkach wynikiem jest jedna liczba, która jest traktowana przez R jako jednoelementowy wektor. Stąd napis `[1]`. Do zagadnienia wyświetlania wektorów jeszcze wrócimy.

Jak widać liczenie w programie R to nic trudnego. Do dyspozycji mamy wszystkie popularne operatory arytmetyczne (ich lista znajduje się w tabeli 1.3). Wyrażenia arytmetyczne można grupować wykorzystując nawiasy (). W programie R dostępne są również popularne funkcje arytmetyczne (ich lista znajduje się w tabeli 1.2) oraz najpopularniejsze funkcje trygonometryczne (wymienione w tabeli 1.1). Z funkcji tych korzysta się intuicyjnie (patrz przykład powyżej). Warto pamiętać, że implementacja tych funkcji często jest bardzo zaawansowana po to, by wyniki numeryczne były wyznaczane z możliwie największą precyzją.

Można też grupować wyrażenia arytmetyczne nawiasami klamrowymi {}. Co prawda R inaczej interpretuje oba typy nawiasów, ale efekt końcowy będzie taki sam.



Warto zwrócić uwagę na funkcje `expm1()` i `log1p()`. Ze względu na ograniczoną możliwość przechowywania i operowania przez procesor na liczbach rzeczywistych, wykonywanie dodawania lub odejmowania na liczbach różniących się o kilka lub kilkanaście rzędów prowadzi do sporych błędów numerycznych. Z tego też powodu w praktycznie każdym kalkulatorze (i również w większości pakietów statystycznych) wartość wyrażenia `1-exp(0.1^15)` jest wyznaczana z błędem względnym rzędu 10%. Podobnie wyrażenie `log(1+0.1^20)` jest wyliczane jako `0` (a więc z błędem względnym wynoszącym 100%). W tych sytuacjach dużo dokładniejsze wyniki będą wyznaczone, gdy użyjemy funkcji `expm1()` i `log1p()`.

```
1-exp(0.1^15)
## [1] -1.110223e-15
expm1(0.1^15)
## [1] 1e-15
log(1+0.1^20)
## [1] 0
log1p(0.1^20)
## [1] 1e-20
```

To jeszcze nie koniec możliwości kalkuRatora. Dostępnych jest znacznie więcej funkcji, które ucieszą każdego inżyniera. Listę bardziej popularnych zamieszczamy w tabeli 1.4. Wybrane bardziej specjalistyczne funkcje w tym: funkcje Bessela, bazy wielomianów ortogonalnych itp. zostaną opisane w kolejnych rozdziałach.

**Tabela 1.1:** Lista funkcji trygonometrycznych z pakietu `base`

| | |
|---|---|
| `cos(x)/sin(x)` | Wartość funkcji cosinus/sinus w punkcie `x`. |
| `tan(x)` | Wartość funkcji tangens w punkcie `x`. |
| `acos(x)/asin(x)` | Wartość funkcji arcus cosinus/sinus w punkcie `x`. |
| `atan(x)` | Wartość funkcji arcus tangens w punkcie `x`. |
| `atan2(y, x)` | Funkcja wyznaczająca kąt (w radianach) pomiędzy osią OX a wektorem o początku w punkcie (0,0) a końcu w punkcie (x,y). Wygodna funkcja do zamiany współrzędnych w układzie kartezjańskich, na współrzędne w układzie biegunowym. |

**Tabela 1.2:** Lista funkcji arytmetycznych z pakietu `base`

| | |
|---|---|
| `round(x)` | Liczba całkowita najbliższa wartości `x`. |
| `signif(x,k)` | Wartość `x` zaokrąglona do `k` miejsc znaczących. |
| `floor(x)` | Podłoga, czyli największa liczba całkowita nie większa od `x`. |
| `ceiling(x)` | Sufit, czyli najmniejsza liczba całkowita nie mniejsza od `x`. |
| `trunc(x)` | Wartość `x` po odcięciu części rzeczywistej, dla liczb dodatnich działa jak `floor()`, dla ujemnych jak `ceiling`. |
| `abs(x)` | Wartość bezwzględna z `x`. |
| `log(x)` | Logarytm naturalny z `x`. |
| `log(x, base)` | Logarytm o podstawie `base` z `x`. |
| `log10(x)` | Logarytm o podstawie `10` z `x`. |
| `log2(x)` | Logarytm o podstawie `2` z `x`. |
| `exp(x)` | Funkcja wykładnicza (eksponenta) z `x`. |
| `expm1(x)` | Funkcja równoważna wyrażeniu `exp(x)-1`, ale wyznaczona z większą dokładnością dla x\|«1\|. |
| `log1p(x)` | Funkcja równoważna wyrażeniu `log(1+x)`, ale wyznaczona z większą dokładnością dla x\|«1\|. |
| `sqrt(x)` | Pierwiastek kwadratowy z `x`, równoważne poleceniu `x^0.5`. |

**Tabela 1.3:** Lista operatorów arytmetycznych

| | |
|---|---|
| `- x` | Zmiana znaku `x`. |
| `x + y (x - y)` | Suma (różnica) `x` i `y`. |
| `x * y (x / y)` | Iloczyn (iloraz) `x` i `y`. |
| `x ^ y` | Liczba `x` do potęgi `y`. |
| `x %% y` | Reszta z dzielenia `x` przez `y` (tzw. dzielenie modulo). |
| `x %/% y` | Część całkowita z dzielenia `x` przez `y`. |

**Tabela 1.4:** Lista funkcji specjalnych i funkcji do operacji na liczbach zespolonych

| | |
|---|---|
| `beta(a,b)` | Wartość funkcji $B(a,b)$ o argumentach `a` i `b`. |
| `lbeta(a,b)` | Wartość logarytmu z funkcji $B(a,b)$. |
| `gamma(x)` | Wartość funkcji $\Gamma(x)$. |
| `lgamma(x)` | Wartość logarytmu z funkcji $\Gamma(x)$. |
| `digamma(x)` | Druga pochodna z logarytmu funkcji $\Gamma(x)$. |
| `trigamma(x)` | Trzecia pochodna z logarytmu funkcji $\Gamma(x)$. |
| `psigamma(x, deriv)` | Pochodna rzędu `deriv` z logarytmu funkcji $\Gamma(x)$. |
| `choose(n,k)` | Liczba kombinacji $k$ elementowych ze zbioru $n$ elementowego. |
| `lchoose(n,k)` | Logarytm z liczby kombinacji $k$ elementowych ze zbioru $n$ elementowego. |
| `combn(n,k)` | Lista wszystkich kombinacji $k$ elementowych ze zbioru $n$ elementowego. |
| `factorial(x)` | Silnia z `x`. |
| `lfactorial(x)` | Logarytm z silni z `x`. |
| `convolve(x,y)` | Splot wektorów `x` i `y`. |
| `complex(real=0,im=0, modulus,argument)` | Konstrukcja liczb zespolonych przez określenie części rzeczywistej i urojonej lub modułu i argumentu. |
| `as.complex(x, ...)` | Konwersja `x` na liczbę zespoloną. |
| `is.complex(x)` | Test, czy argument `x` jest liczbą zespoloną. |
| `Re(x)` | Część rzeczywista liczby zespolonej `x`. |
| `Im(x)` | Część urojona liczby zespolonej `x`. |
| `Mod(x)` | Moduł liczby zespolonej `x`. |
| `Arg(x)` | Argument liczby zespolonej `x`. |
| `Conj(x)` | Sprzężenie liczby zespolonej `x`. |

### 1.5.4 Kilka przykładowych sesji w programie R

W dalszej części tej książki na przykładach pokażemy, co można robić w programie R, na jakich obiektach i w jaki sposób można pracować oraz jakie efekty można uzyskać. W tym podrozdziale nakreślimy wyłącznie kilka ogólnych idei oraz pokażemy kilka przykładowych sesji R, tak by łatwiej było przedzierać się przez późniejsze, sformalizowane opisy. Aby zdobyć biegłość w programowaniu w programie R trzeba ćwiczyć i eksperymentować (tak jak i w nauce każdego języka, czy to języka programowania czy języka naturalnego). Dlatego po przeczytaniu tego podrozdziału warto spróbować samodzielnie napisać kilka programów w programie R. Osoby nie lubiące uczenia się na przykładach powinny ten podrozdział ominąć i przejść do kolejnego.

Zakładam, że czytelnik wie, czym są zmienne i do czego się ich używa. Jeżeli nie, to bez wdawania się w szczegóły może przyjąć, że zmienna reprezentuje pewne wirtualne, nazwane pudełko, w którym możemy przechowywać wartości.

Przykłady rozpocznijmy od operacji na zmiennych.Poniższe przykłady warto samodzielnie uruchomić w programie R. W tym celu należy wpisać zawartość wszystkich linijek nierozpoczynających się od znaku komentarza #.

Zaczynamy od przypisania wartości do zmiennych a i b.

```
a <- 3
b <- 5
```

Przypisać wartość do zmiennej można i operatorem = i operatorem <-. Ale ten pierwszy może występować też w innych kontekstach. Więc do oznaczenia przypisania do zmiennej dla czytelności będziemy używać wyłącznie <-.

Teraz możemy wykonać operacje na tych zmiennych.

```
a + b
## [1] 8
```

Jeżeli nie wiemy, dlaczego na ekranie pojawiła się cyfra 8, to należy rozpocząć lekturę tego rozdziału od początku. Wykonajmy bardziej zaawansowaną operację i wynik przypiszmy do zmiennej c.

```
c <- a/b + 2*b + 1
```

Podając tylko nazwę zmiennej, powodujemy wyświetlenie jej wartości.

```
c
## [1] 11.6
```

Jeżeli przypisanie otoczymy nawiasami, to wymuszamy wypisanie wyniku przypisania.

```
(napis <- "Ala ma kota")
## [1] "Ala ma kota"
```

Bez względu na to, jak zaawansowane analizy będą wykonywane, jednym z efektów, które na pewno pojawi się na ekranie jest komunikat o błędzie. Należy się zawczasu oswoić z reakcją pakietu R na błędy. W podrozdziale 2.9.1 poznamy inne sposoby radzenia sobie z błędami.

Jak pisaliśmy wcześniej program R jest „wrażliwy” na wielkość liter. `napis` i `Napis` to dwie różne zmienne. Zaskakuje to zazwyczaj osoby przyzwyczajone do programów nierozróżniających wielkości liter (np. przyzwyczajonych do programu SAS).

Jedną z najczęstszych przyczyn komunikatu o błędzie, jest użycie nazwy zmiennej, której program R nie rozpoznaje (być może z powodu literówki w nazwie lub złej wielkości liter). W takiej sytuacji sygnalizowany jest następujący komunikat błędu.

```
brakZmiennej + 2
## Error: object "brakZmiennej" not found
```

Błąd pojawi się również przy próbie wywołania nieistniejącej funkcji, jeżeli napotkamy taki błąd, to być możne funkcja, której chcemy użyć jest w pakiecie, który nie został jeszcze załadowany.

```
brakFunkcji()
## Error: could not find function "brakFunkcji"
```

Jeżeli nie podamy wszystkich wymaganych argumentów funkcji, to też możemy się spodziewać błędu.

```
cov(1)
## Error in cov(1) : supply both 'x' and 'y' or a matrix-like 'x'
```

Częstym błędem jest nie dokończenie polecenia, nie zamknięcie nawiasu lub nie zamknięcie łańcucha znaków, w tej sytuacji R sygnalizuje, że czeka na resztę polecenia.

```
lancuch = "
+ "
2 +
+ 2
## [1] 4
```

Możemy też operować na liczbach zespolonych (jednak trzeba to robić z uwagą, patrz poniższy przykład). Lista funkcji do operowania na liczbach zespolonych umieszczona jest w tabeli 1.4.

Jeżeli nie wiemy, co to liczby zespolone, to pomijamy ten przykład.

```
sqrt(-17)
## [1] NaN
## Warning message:
## In sqrt(-17) : NaNs produced
```

Nie tak miało być, jeżeli chcemy korzystać z arytmetyki na liczbach zespolonych, trzeba to jawnie określić.

```
sqrt(-17+0i)
## [1] 0+4.123106i
(2+4i)*(3-2i)
## [1] 14+8i
```

Funkcja `c()` jest bardzo często wykorzystywana do tworzenia wektorów liczb, napisów, wartości logicznych. Skleja ze sobą wartości podane jako jej argumenty.

A teraz skonstruujemy wektor liczb korzystając z funkcji `c()` i wykonamy na nim kilka operacji. Prześledźmy poniższy przykład.

```
(wektor <- c(11, 13, 10.5, -3, 11))
## [1] 11.0 13.0 10.5 -3.0 11.0
```

Na takim wektorze możemy wykonywać operacje arytmetyczne.

```
wektor^2
## [1] 121.00 169.00 110.25   9.00 121.00
1 / wektor
## [1]  0.09090909  0.07692308  0.09523810 -0.33333333  0.09090909
wektor - 2
## [1]  9.0 11.0  8.5 -5.0  9.0
```

Wektory można łączyć w jeszcze większe wektory.

```
c(wektor, 0, 3:5, wektor)
## [1] 11.0 13.0 10.5 -3.0 11.0  0.0  3.0  4.0  5.0  11.0 13.0 10.5 -3.0
    11.0
```

Długie sekwencje liczb łatwiej jest generować używając operatora `:`.

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

Funkcja `rep()` replikuje wektor określoną liczbę razy.

```
rep(1:2, times=5)
##  [1] 1 2 1 2 1 2 1 2 1 2
rep(1:2, each=5)
##  [1] 1 1 1 1 1 2 2 2 2 2
```

Możemy operować na wektorze wartości logicznych (o tym jeszcze będzie).

```
wektor <- c(11, 13, 10.5, -3, 11)
wektor > 0
## [1]  TRUE  TRUE  TRUE FALSE  TRUE
```

W powyższych przykładach wykonaliśmy operacje na całym wektorze. Możemy również manipulować fragmentami wektora lub poszczególnymi elementami

wektora. Poniżej kilka przykładów jak to zrobić. Więcej o tym, jak korzystać z elementów wektora będzie w następnym podrozdziale.

Co jest w pierwszym elemencie wektora `wektor`? A co jest w jego drugim i trzecim elemencie?

```
wektor[1]
## [1] 11
wektor[2:3]
## [1] 13.0 10.5
```

Fragment wektora też jest wektorem możemy więc na nim swobodnie wykonywać dowolne operacje.

```
wektor[2:3] + 4
## [1] 17.0 14.5
```

Co jest w elemencie 1, 3 i 5.

```
wektor[c(1,3,5)]
## [1] 11.0 10.5 11.0
```

Wypiszmy wartości dodatnie z wektora (wartości o indeksach odpowiadającym wartościom dodatnim).

```
wektor[wektor>0]
## [1] 11.0 13.0 10.5 11.0
```

Strukturą bardziej złożoną od wektora jest macierz. W poniższym przykładzie zadeklarujemy macierz o wymiarach $2 \times 3$ i wykonamy na niej kilka operacji arytmetycznych. Zacznijmy od stworzenia macierzy złożonej z samych zer.

```
macierz <- matrix(0,2,3)
macierz
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
```

Tak jak w przypadku wektorów, również na macierzy możemy wykonywać operacja arytmetyczne (jak się niedługo okaże, macierz jest wektorem).

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

A teraz tworzymy macierz, której elementami są kolejne liczby całkowite.

```
macierz <- matrix(1:6,2,3)
macierz
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

Wyświetlmy tylko drugą kolumnę tej macierzy.

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

A teraz drugi wiersz.

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

Z algebry znamy ciekawsze operacje na macierzach. Zobaczmy więc, jak mnożyć macierze, jak liczyć ich wyznaczniki, odwrotności i iloczyny. Zacznijmy od zdefiniowania dwóch macierzy o wymiarach `2x2`.

```
(A <- B <- matrix(1:4,2,2))
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

Pierwsza próba mnożenia, mnożone są elementy macierzy pierwszy z pierwszym, drugi z drugim itp..

```
A * B
##      [,1] [,2]
## [1,]    1    9
## [2,]    4   16
```

Mnożenie macierzowe wykonuje się operatorem `%*%`, wynik jest inny.

```
A %*% B
##      [,1] [,2]
## [1,]    7   15
## [2,]   10   22
```

Policzmy wyznacznik z macierzy A i macierz odwrotną do A.

```
det(A)
## [1] -2
solve(A)
##      [,1] [,2]
## [1,]   -2  1.5
## [2,]    1 -0.5
```

Na koniec wyznaczmy jeszcze wartości własne i wektory własne.

```
eigen(A)
## $values
## [1]  5.3722813 -0.3722813
## $vectors
##            [,1]       [,2]
## [1,] -0.5657675 -0.9093767
## [2,] -0.8245648  0.4159736
```

Uwaga na dokładność numeryczną, wyznacznik poniższej macierzy wynosi zero.

```
(mat <- matrix(c(2, -1, 1, 1, 2, -3, 7, -1, 0),3,3))
##      [,1] [,2] [,3]
## [1,]    2    1    7
## [2,]   -1    2   -1
## [3,]    1   -3    0
det(mat)
## [1] 3.108624e-15
```

To tyle tytułem rozgrzewki, nadszedł czas na trochę teorii.

### 1.5.5 Podstawy składni języka R

Poniżej przedstawimy podstawy składni języka R. Wprowadzimy też takie pojęcia jak typ, obiekt, konwersja itp. Opanowanie tych pojęć i poniżej opisanych informacji jest niezbędne, by móc sprawnie poruszać się po kolejnych rozdziałach.

Obiektowość w programie R jest inna niż obiektowość znana z Javy czy Smalltalka. Więcej uwagi temu tematowi poświęcimy w kolejnym rozdziale.

#### 1.5.5.1 Obiekty

Praktycznie wszystko z czym mamy do czynienia w języku R jest obiektem. Obiekty można podzielić (nie wdając się w formalne szczegóły) na kilka typów (rodzajów) przedstawionych poniżej:

Nie jest to kompletne zestawienie, pomijamy tutaj rzadko używane niskopoziomowe konstrukcje, pełny opis znaleźć można w dokumencie [45]

- **Typ liczbowy.** Obiekty tego typu przechowują liczby, zarówno całkowite jak i rzeczywiste. Wpisując liczby dozwolona jest notacja naukowa (np. `2.5e3` oznacza `2500`). Kropką dziesiętną w programie R jest kropka. Wyróżnioną wartością jest `NaN` (to skrót rozwijający się w ang. *not a number*, czyli „nie liczba"). Ta wartość może pojawić się w wyniku wykonania niepoprawnego działania (np. próby logarytmowania liczby ujemnej). Literały `Inf` i `-Inf` oznaczają plus i minus nieskończoność.

```
1
## [1] 1
1.5
## [1] 1.5
1.5e5
## [1] 150000
```

W tej książce będziemy korzystać przemiennie z różnych nazw dla typu czynnikowego, zdając sobie sprawę, że przez różne grupy użytkowników jest on różnie nazywany. Programiści języków typu C++ i niższego poziomu, przyzwyczajeni są do nazwy typ wyliczeniowy. Nazwa typ kategoryczny bierze się z nazywania możliwych wartości zmiennej danego typu kategoriami. Nazwa typ czynnikowy jest najczęściej używana wśród statystyków, gdzie możliwe wartości odpowiadają różnym poziomom pewnego czynnika.

- **Typ czynnikowy (nazywany również wyliczeniowym lub kategorycznym).** Ten typ jest przydatny do przechowywania wektorów wartości występujących na kilku poziomach (w kilku kategoriach). Przykładowo płeć występuje na dwóch poziomach, tzn. może przyjmować tylko dwie wartości, dlatego przechowując w programie R wektor danych opisujących płeć, najlepiej użyć typu czynnikowego. Zmienne tego typu są najczęściej wykorzystywane do definiowania grup. Zmienne typu czynikowego zajmują mniej miejsca w pamięci niż odpowiadające im łańcuchy znaków. Wewnętrznie takie wektory przechowywane są jako wektory liczb przez co można na nich szybciej wykonywać określone funkcje. Gdy możemy, warto używać tego typu dla poprawienia efektywności. Ponadto wiele funkcji programu R (szczególnie statystycznych) jest w stanie rozpoznać, że argument jest typu wyliczeniowego i zastosować odpowiednie działania, np. wyznaczyć liczebności poszczególnych grup, zastosować kodowanie z użyciem zmiennych pustych itp.

  Zazwyczaj zmienne tego typu tworzy się z użyciem funkcji `factor()`. Na poniższym przykładzie konstruujemy czteroelementowy wektor elementów typu wyliczeniowego, ze słownikiem składającym się z dwóch wartości.

```
(nz <- factor(c("sierzant", "kapitan", "sierzant", "sierzant")))
## [1] sierzant kapitan  sierzant sierzant
## Levels: kapitan sierzant
```

Zobaczmy podsumowanie tego wektora.

```
summary(nz)
##  kapitan sierzant
##        1        3
```

- **Typ znakowy.** Wartościami obiektów tego typu są napisy (będziemy też używać nazwy łańcuchy znaków). W programie R napisy rozpoczynane są znakiem ' lub " oraz kończone takim samym znakiem. W łańcuchu znaków mogą występować dowolne znaki w tym znaki specjalne rozpoczynające się od znaku \. Wybrane znaki specjalne to: `\n` – znak nowej linii, `\t` – znak tabulacji, \\ – oznaczający znak \, znak \" – oznaczający ", itp.

  Z łańcuchów znaków można wycinać fragmenty, sklejać, wyszukiwać podciągi znaków i wykonywać wiele innych operacji, o których napiszemy w kolejnych podrozdziałach.

```
"To jest napis"
## [1] "To jest napis"
'To tez jest napis'
## [1] "To tez jest napis"
"To jest napis 'a to jest napis wewnetrzny'"
## [1] "To jest napis 'a to jest napis wewnetrzny'"
```

  Funkcja `cat()`' wyświetla napis w sposób niesformatowany.

```
cat(" co \t to \\ teraz\"\n\n bedzie?")
##  co      to \ teraz"
##
##  bedzie?
```

  Napisy można sklejać.

```
paste("Napis", "napis doklejony", 12)
## [1] "Napis napis doklejony 12"
```

- **Typ logiczny.** Obiekty tego typu przechowują jedną z dwóch wartości, logiczną prawdę (oznaczaną przez literał `TRUE` lub jego skrót `T`) albo logiczny fałsz (oznaczany przez literał `FALSE` lub jego skrót `F`). Na tych obiektach można wykonywać operacje logiczne oraz arytmetyczne (o tym w kolejnych podrozdziałach).

  Jeżeli wartości logiczne znajdą się w wyrażeniu arytmetycznym, to zostaną skonwertowane na liczby, odpowiednio, 1 i 0.

Literały TRUE i FALSE to słowa zastrzeżone, ich wartości nie mogą być zmodyfikowane. Natomiast T i F to zwykłe zmienne, których wartość można dowolnie modyfikować.

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

  Testowanie równości.

```
1==2
## [1] FALSE
2==2
## [1] TRUE
```

Wyrażenie arytmetyczne, następuje automatyczna konwersja wartości typu logicznego na liczbę.

```
(2==2) + 2
## [1] 3
```

Wyrażenie logiczne, w użyciu operatory sumy logicznej i negacji.

```
(1==0) | !(1==0)
## [1] TRUE
```

I don't like to see the use of c() for its side effects. In this case Marc's as.vector seems to me to be self-explanatory, and that is a virtue in programming that is too often undervalued.

Brian D. Ripley (on how to convert a matrix into a vector) fortune(185)

- **Wektor elementów.** Wektor to uporządkowany zbiór obiektów tego samego typu. Do tworzenia wektora z pojedynczych elementów lub innych wektorów służy funkcja `c()`. W programie R nie ma rozróżnienia na pojedynczą wartość i wektor, pojedyncze wartości traktowane są jako jednoelementowe wektory. Poniżej przedstawiamy konstrukcje wektora składającego się z trzech liczb oraz wektora – sekwencji 30 liczb. Tak długie wektory wyświetlane są w kilku wierszach. Kolejne wiersze rozpoczynają się od, otoczonego nawiasami kwadratowymi, indeksu pierwszego elementu wyświetlonego w danej linii.

```
c(1, 3, 4)
## [1] 1 3 4
(1:30)*2
##  [1]  2  4  6  8 10 12 14 16 18 20
## [11] 22 24 26 28 30 32 34 36 38 40
## [21] 42 44 46 48 50 52 54 56 58 60
```

Elementy wektora mogą mieć nazwy.

```
c(pierwszy = 12, drugi = 10, trzeci = 18)
## pierwszy    drugi   trzeci
##       12       10       18
```

Ponieważ analizując dane często określone transformacje wykonuje się na wszystkich elementach wektora, dlatego też język R został tak zaprojektowany, by operacje na wektorach były możliwie najefektywniejsze. Jeżeli chcemy dodać dwa wektory do siebie wystarczy użyć operatora +. Dodawanie wektorów za pomocą pętli, dodającej element po elemencie będzie znacznie mniej efektywne.

Wszystkie elementy wektora muszą mieć ten sam typ. Wyjątkiem jest umieszczanie w wektorze dowolnego typu wartości `NA` (ang. *not avaliable*) oznaczającej brak wartości. Wykonywanie działań arytmetycznych na wartości `NA` daje w wyniku również wartość `NA`. Niektóre funkcje mają możliwość podania argumentu `na.rm`, który ustawiony na `TRUE` wymusza usuwanie brakujących obserwacji przed kontynuowaniem obliczeń.

Funkcja `mean()` liczy średnią arytmetyczną, ale co ma zrobić z brakującą obserwacją?

```
wektor <- c(1, 2, NA, 40, 51)
mean(wektor)
## [1] NA
```

Dodanie argumentu `na.rm=TRUE` rozwiązuje problem.

```
mean(wektor, na.rm=TRUE)
## [1] 23.5
```

Jeżeli chcemy usunąć z wektora wartości brakujące, to możemy posłużyć się funkcją `na.omit()` (jej wynikiem jest wektor bez elementów NA) lub funkcją `complete.cases()` (jej wynikiem jest wektor wartości logicznych, `TRUE` gdy nie ma `NA` lub `FALSE` gdy jest). Argumentami obu funkcji mogą być wektory, macierze lub ramki danych. Jeżeli argumentem jest ramka danych, to funkcja `na.omit()` usunie cały wiersz, w którym znajdują się brakujące obserwacje, a funkcja `complete.cases()` określi dla każdego wiersza, czy znajdują się w nim brakujące obserwacje. Związana z wartościami brakującymi jest również funkcja `na.fail()` generująca błąd, jeżeli w argumencie tej funkcji znajdują się brakujące obserwacje.

- **Lista.** Podobnie jak wektor, lista to również uporządkowany zbiór elementów. W przeciwieństwie do wektora, elementy listy mogą mieć różne typy. Podobnie jak w przypadku wektora poszczególne elementy mogą mieć nazwy. Listy tworzy się zazwyczaj z użyciem funkcji `list()`. Do elementów listy możemy się odwoływać jak do elementów wektora, korzystając z nazw poszczególnych pól lub z operatora `[[]]`. W poniższym przykładzie konstruujemy listę czterech obiektów różnych typów.

```
list(imie=c("Jan","Tomasz"), nazwisko="Kowalski", wiek=25,
    czyWZwiazku=T)
## $imie
## [1] "Jan"    "Tomasz"
##
## $nazwisko
## [1] "Kowalski"
##
## $wiek
## [1] 25
##
## $czyWZwiazku
## [1] TRUE
```

- **Macierz.** Macierze tworzy się zazwyczaj funkcją `matrix()`. Parametrami tej funkcji jest wektor inicjujący zawartość macierzy oraz dwie liczby określające wymiary macierzy. Macierz może składać się z liczb, napisów lub wartości logicznych. W poniższym przykładzie konstruujemy macierz o wymiarach `4x2` wypełnioną zerami.

```
matrix(0,2,4)
##      [,1] [,2] [,3] [,4]
## [1,]    0    0    0    0
## [2,]    0    0    0    0
```

Funkcją `array()` można konstruować macierze o większej liczbie wymiarów. Ten temat poruszymy w kolejnych podrozdziałach.

- **Ramka danych.** Szczególnym typem jest ramka danych, nazywana również tabelą danych. Ramka danych jest zazwyczaj kojarzona z macierzową/tabelaryczną strukturą, której elementy w każdej kolumnie są tego samego typu, ale mogą różnić się typami pomiędzy kolumnami. Z tego powodu ramkę danych można traktować jak listę wektorów o tej samej długości, każdy wektor odpowiada jednej kolumnie.

  Ramki danych tworzy się zazwyczaj funkcją `data.frame()`. Poniżej konstruujemy ramkę danych składającą się z trzech trzyelementowych zmiennych.

  Konstruujemy ramkę danych podając wartości dla każdej z kolumn.

```
(ramka <- data.frame(id = c(100,101,102),
                     wiek = c(25,21,22),
                     czy.chlopiec = c(TRUE,TRUE,FALSE)))
##    id wiek czy.chlopiec
## 1 100   25         TRUE
## 2 101   21         TRUE
## 3 102   22        FALSE
```

  Do elementów ramki danych możemy odwoływać się tak, jak do elementów macierzy, a także tak jak do elementów list.

  Dwa różne sposoby odwołania się do drugiej kolumny.

```
ramka$wiek
## [1] 25 21 22
ramka[,"wiek"]
## [1] 25 21 22
ramka[,2]
## [1] 25 21 22
```

- **Typ funkcyjny.** Do konstrukcji obiektów tego typu wykorzystuje się słowo kluczowe `function`. Więcej o funkcjach, w tym o pisaniu własnych funkcji, znaleźć można w podrozdziale 1.6.2.

#### 1.5.5.2 Konwersje

Typ zmiennej nie jest przypisany do zmiennej na stałe. Możemy zmieniać typy zmiennej nie podając nowej wartości dla zmiennej licząc na automatyczną zmianę wartości z jednego typu na drugi. Proces zmiany typu nazywamy konwersją typu. Najczęstsze konwersje to zamiana na typ znakowy (funkcja `as.character()`) lub na typ liczbowy (funkcja `as.numeric()`).

Konwertować można pojedyncze wartości jak również złożone struktury takie jak lista lub macierz. W przypadku konwersji struktury konwertowany jest każdy element tej struktury (listy, macierzy itp.) lub też konwertowana jest cała struktura, np. lista może być zamieniana na wektor. Lista popularnych funkcji konwertujących lub sprawdzających typ zmiennej jest przedstawiona w tabeli 1.5.

Konwertując obiekty typu wyliczeniowego na typ liczbowy należy być ostrożnym, aby nie być zaskoczonym wynikiem takiej konwersji. W przykładzie poniżej widzimy, co złego może się stać przy nieostrożnym konwertowaniu liczb.

**Tabela 1.5:** Funkcje pozwalające na sprawdzenie lub konwersję typu zmiennej

| | |
|---|---|
| `is.numeric()` | Test czy argument jest liczbą. |
| `is.integer()` | Test czy argument jest liczbą całkowitą. |
| `is.double()` | Test czy argument jest liczbą rzeczywistą. |
| `is.complex()` | Test czy argument jest liczbą zespoloną. |
| `is.logical()` | Test czy argument jest wartością logiczną. |
| `is.character()` | Test czy argument jest znakiem lub napisem. |
| `is.factor()` | Test czy argument jest typu wyliczeniowego. |
| `is.na()` | Test czy argument jest wart. nieokreśloną (`NA`). |
| `is.nan()` | Test czy argument jest niewłaściwą liczbą (`NaN`). |
| `as.numeric()` | Konwersja na wartość liczbową. |
| `as.integer()` | Konwersja na wartość całkowitoliczbową. |
| `as.double()` | Konwersja na wartość rzeczywistą. |
| `as.complex()` | Konwersja do liczby zespolonej. |
| `as.logical()` | Konwersja na wartość logiczną. |
| `as.character()` | Konwersja na typ znakowy. |
| `as.factor()` | Konwersja na typ wyliczeniowy. |
| `as.list()` | Konwersja do listy. |
| `unlist()` | Konwersja z listy do wektora. |
| `as.matrix()` | Konwersja na macierz. |
| `as.data.frame()` | Konwersja na ramkę danych. |
| `class()`, `mode()` | Odczytanie klasy / typu argumentu. |

```
(wektor <- factor(c(-2,2)))
## [1] -2 2
## Levels: -2 2
```

Nie takiego wyniku się spodziewaliśmy.

```
as.numeric(wektor)
## [1] 1 2
```

Konwersja typu `factor` na `numeric` polega na tym, że kolejnym poziomom przypisywane są kolejne liczby naturalne. Stąd w powyższym przykładzie wynik `1 2`. Bardziej oczekiwany wynik uzyskamy konwertując „po drodze" wektor na typ znakowy.

```
as.character(wektor)
## [1] "-2" "2"
```

Tak miało być.

```
as.numeric(as.character(wektor))
## [1] -2  2
```

Jeżeli nie jesteśmy pewni jakiego typu jest zmienna, to możemy jej typ sprawdzić funkcją `class()` lub `mode()`. Możemy też wykorzystać funkcje z tabeli 1.5. Proszę zwrócić uwagę, że dwie przedstawione w tej tabeli funkcje: `is.na()` i `is.nan()` testują wartości, a nie typ.

```
wektor <- 1:6
class(wektor)
## [1] "integer"
```

Tryb przechowywania elementów wektora to tryb liczbowy.

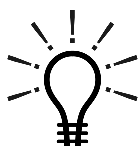
```
mode(wektor)
## [1] "numeric"
is.numeric(wektor)
## [1] TRUE
```

Ten wektor nie jest wektorem znaków.

```
is.character(wektor)
## [1] FALSE
```

I żadna z jego wartości nie jest NA.

```
is.na(wektor)
## [1] FALSE FALSE FALSE FALSE FALSE FALSE
```



Klasę obiektu „widzianą" przez funkcję `class()` można dowolnie zmieniać, możemy do obiektu przypisywać wiele klas o dowolnych nazwach, o czym jeszcze napiszemy więcej w kolejnych rozdziałach. Funkcja `mode()` informuje o wewnętrznej reprezentacji danej zmiennej i nie możemy na wynik tej funkcji bezpośrednio wpływać. Więcej o obu tych funkcjach przeczytać można w podrozdziale 2.1.7.

#### 1.5.5.3 Zmienne

Zmienne służą do przechowywania wprowadzonych danych lub wyników wykonanych poleceń. Najczęściej te wartości chcemy przechować, aby móc je później ponownie wykorzystać. Do wartości zmiennych odwołujemy się podając nazwę zmiennej. Nazwa zmiennej powinna rozpoczynać się literą, może składać się z liter, cyfr i kropek. Istotna jest wielkość liter, więc zmienne `x` i `X` to dwie różne zmienne.

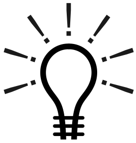
Do zmiennej można przypisać wartość dowolnym z następujących operatorów przypisania (jeszcze inne, mniej popularne sposoby, są opisane na końcu tego podrozdziału):

- operator `=`, przypisuje wartość znajdującą się z prawej strony do zmiennej znajdującej się po lewej stronie,
- operator `<-`, działa tak samo jak powyższy ale ma wyższy priorytet,
- operator `->`, przypisuje wartość znajdującą się z lewej strony do zmiennej znajdującej się po prawej stronie.

Wartość do zmiennej można też przypisać funkcją `assign()`, ale korzystanie z powyższych operatorów jest wygodniejsze. Poniżej przedstawiamy różne sposoby przypisywania wartości do zmiennych.

```
c(13, 13) -> zmienna.z.kropka
imieN <- "Ola"
i2 = 4
assign("zmienna",14)
```

Do przypisania wartości można używać również operatorów ->> i <<-, które mają podobne działanie do operatorów -> i <-, które przypisują wartości do zmiennych w najwyższej przestrzeni nazw, w której dana zmienna jest już zdefiniowana. Więcej o tych dwóch operatorach napiszemy w kolejnych rozdziałach.



Jak napisaliśmy wcześniej wszystko jest obiektem, nawet funkcja. Operator jest również funkcją a tym samym jest również obiektem, tyle że składnia języka R pozwala na jego specjalny zapis. I tak operator = jest w rzeczywistości funkcją dwuargumentową `` `=`() ``. Oznacza to, że zapis `x = 3` jest równoważny zapisowi `` `=`(x, 3) ``. Oczywiście ten pierwszy zapis jest dla człowieka bardziej czytelny. Podobnie jest z pozostałymi operatorami.

Jeżeli chcemy, by po przypisaniu wartość tego przypisania została wyświetlona na ekranie, to operacje przypisania należy zamknąć w okrągłych nawiasach.

```
(zmienna <- 2^10)
## [1] 1024
```

Warto wspomnieć w tym miejscu o zmiennej `.Last.value`. Przechowuje ona wynik ostatnio wykonanego polecenia. Ta zmienna może być bardzo przydatna, jeżeli wykonaliśmy jakieś długo liczące się polecenie, a nie zapisaliśmy jego wyniku. W zmiennej `.Last.value` będzie ta wartość.

W większości zastosowań operatory = i <- można stosować zamiennie. Jednak nie zawsze mają one to samo działanie. Różnica pojawia się np. gdy operatory te użyte są przy określaniu argumentu funkcji. Operator = służy do wskazania, który argument funkcji określamy, operator <- zachowuje się jak zwykły operator przypisania (jeszcze do tego wrócimy).
Kolejna różnica polega na tym, że operator <- ma wyższy priorytet. Prześledźmy poniższy kod.

Do zmiennej `base` przypisujemy wartości liczbowe, wynik przypisania podawany jest jako argument do funkcji.

```
log(base <- 100, base <- 10)
## [1] 2
```

A teraz używamy operatora =, który wskazuje argument. Nie można dwa razy wskazać tego samego argumentu, stąd komunikat błędu.

```
log(base = 100, base = 10)
## Error: formal argument "base" matched by multiple actual arguments
```

Co z kolejnością wiązania operatora? Z poniższego przykładu wynika, że <- jest wykonywane przed =.

```
a = b = 5
a <- b <- 5
a = b <- 5
a <- b = 5
## Error in (a <- b) = 5 : could not find function "<-<-"
```

#### 1.5.5.4 Indeksy

Do elementów składowych wektorów, list, macierzy i ramek danych możemy się odwoływać na różne sposoby. Przedstawiamy te sposoby poniżej, wraz z krótkim opisem. Więcej szczegółowych informacji znaleźć można wpisując w programie R polecenie `?Extract`.

- W notacji wektorowej: `zmienna[zakres]`.

  W tym przypadku `zmienna` jest listą, wektorem, macierzą lub ramką danych, `zakres` jest wektorem liczb całkowitych lub jedną liczbą całkowitą. Wynikiem jest lista(wektor) zawierająca wybrane elementy.

  W przypadku indeksowania list, ramek danych lub wektorów z nazwanymi elementami, możemy w nawiasach `[]` podać wektor nazw elementów. W tym przypadku wybrane zostaną elementy o nazwach wskazanych przez wektor indeksów—nazw.

  Jeżeli `zakres` jest wektorem liczb ujemnych, to zwrócone będą wszystkie elementy z listy(wektora) **poza** wskazanymi pozycjami. Nie można w `zakres` mieszać indeksów dodatnich i ujemnych.

  ```
  wektor <- 1:10
  wektor
  ## [1]  1  2  3  4  5  6  7  8  9 10
  wektor[1:3]
  ## [1] 1 2 3
  wektor[c(-1,-3,-5)]
  ## [1]  2  4  6  7  8  9 10
  ```

  Indeksy nie muszą być liczbami, mogą być nazwami elementów.

  ```
  (wektor <- c(a=1, b=2, c=3))
  ## a b c
  ## 1 2 3
  wektor[c("a","c")]
  ## a c
  ## 1 3
  ```

- W notacji macierzowej: `zmienna[zakres1,zakres2]`.

  W tym przykładzie `zmienna` jest macierzą lub ramką danych. Jako wynik wybierana jest podmacierz (ramka danych) o wskazanych indeksach. Jeżeli któryś z zakresów nie będzie podany, to w wyniku zostaną wybrane wszystkie elementy w danym wierszu/kolumnie.

  ```
  macierz <- matrix(1:4,2,2)
  macierz
  ##      [,1] [,2]
  ## [1,]    1    3
  ## [2,]    2    4
  ```

  Wybieramy tylko pierwszy wiersz

  ```
  macierz[1,]
  ## [1] 1 3
  ```

Tylko elementy poza pierwszym wierszem i pierwszą kolumną.

```
macierz[-1,-1]
## [1] 4
```

Domyślnie, jeżeli wynikiem ma być pojedynczy wiersz lub kolumna, to „gubiony" jest wymiar macierzy, wynik staje się wtedy wektorem a nie macierzą. Jeżeli chcemy zagwarantować, że wynik będzie macierzą, chociażby po to by odwoływać się do jej wartości jak do elementów macierzy, to przy indeksowaniu należy dodać argument `drop=TRUE`. Poniżej przedstawiamy przykład ilustrujący tę różnicę.

Tak, to kolejny przykład, że operator indeksowania jest w rzeczywistości funkcją, a drop jest jednym z jej argumentów. Ale dla czytelności traktujemy operator indeksowania w sposób specjalny

Wynik poniższych instrukcji jest wektorem

```
macierz[1,]
## [1] 1 3
macierz[1,,drop=TRUE]
## [1] 1 3
```

Ale tej jest już macierzą.

```
macierz[1,,drop=FALSE]
##      [,1] [,2]
## [1,]    1    3
```

- W notacji listowej: `zmienna$wartosc1`

  W tym przykładzie `zmienna` to lista, wektor lub ramka danych. Wynikiem zastosowania tego operatora jest element listy/wektora (lub kolumna z ramki danych) o nazwie `wartosc1`.

```
osobnik <- list(imie=c("Jan","Tomasz"), nazwisko="Kowalski", wiek
    =25, wzwiazku=TRUE)
osobnik$name
## [1] "Jan"    "Tomasz"

osobnik[[2]]
## [1] "Kowalski"
```

- W notacji nawiasowej: `zmienna[[indeks1]]`.

  W tym przykładzie `zmienna` to wektor, lista, macierz lub ramka danych. Wybierany jest jeden element o indeksie `indeks1` (indeks musi być pojedynczą liczbą, nie może być wektorem). Najczęściej ten sposób indeksowania wykorzystywany jest dla list. Pamiętajmy, że ramka danych też jest listą, dlatego użycie tego sposobu indeksowania na ramce danych spowoduje wybranie wskazanej kolumny ramki danych, podczas gdy użycie tego indeksowania na macierzy spowoduje wybranie jednej wartości z macierzy.

```
macierz <- matrix(1:4,2,2)
```

Macierz jest tak naprawdę wektorem, więc drugi element macierzy to jedna liczba.

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

Pierwszy element listy, wektor dwuelementowy.

```
osobnik <- list(name=c("Jan","Tomasz"), surname="Kowalski", age=25,
    married=TRUE)
osobnik[[1]]
## [1] "Jan"    "Tomasz"
```

Należy bardzo uważać, jeżeli wartość indeksu jest wyliczana wyrażeniem arytmetycznym. Jeżeli indeks będzie wartością niecałkowitą, zostanie on zaokrąglony w dół, co czasem może być przyczyną bardzo trudnych do wykrycia błędów.
W przykładzie poniżej indeks wektora jest wyznaczany jako wyrażenie $(0.1+0.7)*10$, które z uwagi a precyzje obliczeń zostaje wyliczone jako $8 - 8.881784 * 10^{-16}$, a więc „trochę" mniej niż 8. Wyświetlając w konsoli programu R wartość $(0.1 + 0.7) * 10$ zobaczymy jako wynik liczbę 8, ponieważ przy wyświetlaniu liczby są zaokrąglane z dokładnością do pierwszych siedmiu cyfr znaczących. Pomimo tego, jeżeli użyjemy tej wartości do indeksowania wektora, to jako wynik zwrócony będzie element o indeksie 7, bowiem wartości indeksów zaokrąglane są w dół.
Błędy takie jak poniżej przedstawiony bardzo trudno wykryć. Należy więc unikać operacji na liczbach niecałkowitych w indeksach wektorów, a jeżeli już jest to konieczne, warto zatroszczyć się o poprawne zaokrąglanie, np. używając funkcji `round()`.

```
(wektor <- 1:10)
## [1]  1  2  3  4  5  6  7  8  9 10
(0.1+0.7)*10
## 8
wektor[ (0.1+0.7)*10 ]
# [1] 7
```

Ups... to wcale nie jest ósmy element.

```
wektor[ round((0.1+0.7)*10) ]
## [1] 8
```

Przydatną funkcją do operacji na wektorach wartości logicznych jest funkcja `which()`. Jej wynikiem są indeksy elementów na których w wektorze wejściowym występowała wartość `TRUE`. Jako argument tej funkcji możemy podać warunek logiczny i jako wynik otrzymamy indeksy elementów spełniających ten warunek. Jeżeli interesują nas indeksy wystąpień pewnego wektora elementów w innym wektorze możemy użyć funkcji `match()`. Działa ona znacznie szybciej niż odpowiednia instrukcja zapisana z użyciem funkcji `which()`.

Podobne działanie do funkcji `match()` ma binarny operator `%in%`. Jego wynikiem jest wektor wartości logicznych określających, czy jakikolwiek element argumentu prawego wystąpił w danym elemencie argumentu lewego (patrz poniższy przykład). Jeżeli chcemy znaleźć indeks elementu minimalnego lub maksymalnego w wektorze, to możemy skorzystać z funkcji `which.min()`, `which.max()`. Ich wynikami są indeksy pierwszego ekstremum.

```
wektor <- c(11, 13, 10.5, -3, 11, -3)
which(wektor==10.5)
## [1] 3
match(c(11,10.5,-3), wektor)
## [1] 1 3 4
which(wektor<11)
## [1] 3 4 6
which(wektor == min(wektor))  # wszystkie najmniejsze
## [1] 4 6
which.max(wektor)             # jeden najwiekszy
## [1] 2
which.min(wektor)             # jeden najmniejszy
## [1] 4
```

Zmienna `LETTERS` to 26-elementowy wektor kolejnych dużych liter. Na której pozycji są `A`, `Z`, `M` i `G`?

```
LETTERS %in% c("A", "Z", "M", "G")
##  [1]  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
## [11] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [21] FALSE FALSE FALSE FALSE FALSE  TRUE
```

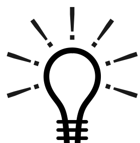
Zauważmy, że kolejność wyznaczonych pozycji nie odpowiada kolejności w jakiej litery były podane. Zbiory nie mają kolejności.

```
which(LETTERS %in% c("A", "Z", "M", "G") )
## [1]  1  7 13 26
```

#### 1.5.5.5 Operatory

W tabeli 1.6 przedstawiono najpopularniejsze operatory w języku R. Z każdego z tych operatorów korzystać można również jak z funkcji, co jest zazwyczaj mniej wygodne w zapisie. Np. wyrażenie `2+3` jest równoważne wyrażeniu `"+"(2,3)` oraz `sum(2,3)`. Można również definiować własne operatory, co opiszemy w podrozdziale 1.6.2.5.

Operatory `&` i | (logiczny iloczyn i logiczna suma) służą do wykonywania operacji na listach lub wektorach, podczas gdy `&&` i || na pojedynczych wartościach.



Logiczna suma odpowiada łącznikowi *lub* a logiczny iloczyn łącznikowi *i* w języku polskim. Jeżeli ten komentarz dużo Ci nie rozjaśnił to znaczy, że nie potrzebujesz korzystać z tych operatorów i się nie przejmuj.

Z uwagi na ochronę danych osobowych, imiona studentek w tym przykładzie zostały zmienione.

Tworzymy dwa wektory wartości logicznych, których elementy mają dodatkowow wektor nazw. Następnie wykonujemy „logiczne i" dla kolejnych par elementów obu wektorów.

```
lubie.statystyke   <- c(ala=FALSE, ola=TRUE, ewa=TRUE)
lubie.prowadzacego <- c(ala=TRUE, ola=TRUE, ewa=FALSE)
lubie.statystyke & lubie.prowadzacego
##   ala   ola   ewa
## FALSE  TRUE FALSE
```

**Tabela 1.6:** Lista operatorów logicznych i arytmetycznych

| | |
|---|---|
| `+, - , *, /, ^` | Operatory dwuargumentowe. Oba argumenty powinny mieć taki sam wymiar lub jeden powinien być wielokrotnością drugiego. |
| `%x%, %*%` | Iloczyn Kroneckera i iloczyn macierzowy dwóch macierzy. |
| `%%, %/%` | Reszta modulo z dzielenia i dzielenie całkowite. |
| <, ==, >, <=, >= | Standardowe operatory porównywania wartości liczbowych. |
| `!, !=` | Operator negacji logicznej i nierówności. |
| &, &&, \|, \|\| | Logiczny iloczyn oraz logiczna suma. |
| `any(), all()` | Logiczna suma(iloczyn) wszystkich elementów wektora. |

A teraz „logiczne i" dla wszystkich elementów obu wektorów.

```
lubie.statystyke && lubie.prowadzacego
## [1] FALSE
```

I jeszcze „logiczne lub" dla kolejnych par elementów obu wektorów.

```
lubie.statystyke | lubie.prowadzacego
##  ala  ola  ewa
## TRUE TRUE TRUE
```

Teraz „logiczne lub" dla wszystkich elementów obu wektorów.

```
lubie.statystyke || lubie.prowadzacego
## [1] TRUE
any(lubie.statystyke)
## [1] TRUE
```

Pomiędzy operatorami || (&&) a | (&) istnieje jeszcze jedna różnica. Operatory | i & zachowują się jakby stosowały tak zwane „gorliwe wartościowanie", czyli wyznaczają wartości wszystkich argumentów tych operatorów a następnie wyznaczają logiczną sumę lub iloczyn. Operatory || i && zachowują się, jakby stosowały tak zwane „leniwe wartościowanie", czyli wyznaczają wartość drugiego argumentu, tylko jeżeli jest ona niezbędna do określenia wartości wyniku. W sytuacji, gdy pierwszy argument operatora || ma wartość `TRUE` (a dla operatora && wartość `FALSE`) nie jest wyliczany drugi argument, ponieważ wynik operatora jest już znany. Warto pamiętać o tych różnicach, często są one źródłem błędów trudnych do wykrycia.

```
wypiszNaEkran <- function(imie) {
    cat(paste(imie, "\n"));   TRUE
}
wypiszNaEkran("ala") | wypiszNaEkran("wylacz komputer")
## ala
## wylacz komputer
## [1] TRUE
wypiszNaEkran("ala") || wypiszNaEkran("ola")
## ala
## [1] TRUE
```

#### 1.5.5.6 Sekwencje

Sekwencje to wektory liczb całkowitych, takie że dwie sąsiednie wartości są oddalone o tą samą wartość, najczęściej o jeden. Takie wektory można generować używając operatora `:` lub funkcji `seq()`. Kilka sposobów generowania sekwencji przedstawimy poniżej.

Operator `:`, generuje sekwencję liczb od ... od ... z krokiem 1.

```
-2:2
## [1] -2 -1  0  1  2
2:-2
## [1] 2 1  0  -1  -2
```

Poniższe wywołanie funkcji `seq()` równoważne jest wywołaniu 1:10.

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

Podajemy zakres wektora, równoważne z użyciem 10:25.

```
seq(10, 25)
##  [1] 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
```

Dodatkowo określamy krok, czyli o ile zwiększane są kolejne wartości.

```
seq(10,25,by=10)
## [1] 10 20
```

Możemy też określić pożądaną długość wektora, funkcja `seq()`' sama zatroszczy się o wybór kroku.

```
seq(10, 25, length.out=10)
##  [1] 10.00000 11.66667 13.33333 15.00000 16.66667 18.33333 20.00000
##  [8] 21.66667 23.33333 25.00000
```

Przydatną funkcją do operowania na sekwencjach liczb (a także na zwykłych wektorach) jest funkcja `sample()`. Losuje ona *k*-elementowy podzbiór (*k* to drugi argument tej funkcji) z wektora danego jako pierwszy argument tej funkcji. Można losować elementy ze zwracaniem (gdy trzeci argument `replace=TRUE`) lub bez zwracania (gdy argument `replace=FALSE`, ustawienie domyślne). Można też wskazać wektor prawdopodobieństw (argument `prob`) określający prawdopodobieństwa wylosowania poszczególnych elementów wektora. W poniższym przykładzie losowany jest dziesięcioelementowy wektor liter. Wykorzystano w tym przykładzie predefiniowany wektor `letters`, czyli wektor małych liter z alfabetu łacińskiego (inne ciekawe predefiniowane wektory to `LETTERS` – duże litery, `month.name` – nazwy miesięcy i `month.abb` – trzyliterowe skróty nazw miesięcy).

Wylosujmy dziesięć liter (losujemy z powtórzeniami).

```
sample(letters, 10, TRUE)
##  [1] "u" "q" "x" "s" "q" "f" "c" "f" "l" "x"
```

Wylosujmy wektor ze zbioru liczb całkowitych od 1 do 3, losując z zadanymi prawdopodobieństwami.

```
sample(1:3, 20, TRUE, prob=c(0.6,0.3,0.1))
##  [1] 2 3 1 1 3 3 1 1 1 1 1 2 1 1 1 2 1 1 2 2
```

#### 1.5.5.7 Komentarze

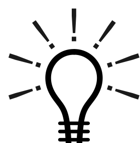
Język R, jak każdy przyzwoity (i wiele nieprzyzwoitych) języków programowania, umożliwia komentowanie fragmentów kodu. Znakiem rozpoczęcia komentarza jest #. Analizując skrypt R interpreter ignoruje ten znak i wszystkie po nim występujące aż do znaku nowej linii (czyli do końca linii).

### 1.5.6 Wyświetlanie i formatowanie obiektów

Dwie funkcje wykorzystywane do wyświetlania obiektów to: `cat()` i `print()`. Obie wyświetlają wartość obiektu, ale każda w odmienny sposób. Aby je porównać zacznijmy od przykładu, w którym wyświetlimy wektor sześciu napisów.

```
nap <- rep(c("Ala ma kota", "Ola nie ma kota", "Ela chce miec kota"),2)
print(nap)
## [1] "Ala ma kota"        "Ola nie ma kota"    "Ela chce miec kota"
## [4] "Ala ma kota"        "Ola nie ma kota"    "Ela chce miec kota"
cat(nap)
## Ala ma kota Ola nie ma kota Ela chce miec kota Ala ma kota Ola nie ma
    kota Ela chce miec kota
```

Ten sam wektor został inaczej wyświetlony przez każdą z tych funkcji. Funkcja `print()` wyświetliła wektor w dwóch liniach (ponieważ w jednej się nie zmieścił), na początku każdej linii zaznaczyła, który element wektora rozpoczyna tę linię. Wyświetlone wartości są w cudzysłowach, dzięki czemu można rozpoznać ich typ. Funkcja `cat()` wyświetliła cały wektor w jednej linii, bez żadnego dodatkowego formatowania.



Domyślnie, jeżeli w wyniku wykonania polecenia w programie R zostanie zwrócona wartość, która nie zostanie przypisana do zmiennej, to wartość ta jest wyświetlana z użyciem funkcji `print()`.

Tak dzieje się zarówno dla prostych wyrażeń arytmetycznych, jak i dla bardziej skomplikowanych obiektów będących wynikami np. funkcji statystycznych (patrz wyniki testów statystycznych). Aby nasz kod był elastyczny, to oprogramowując pewną funkcjonalność, której wynik ma być specyficznie wyświetlony, powinniśmy tę funkcjonalność rozbić na dwie funkcje. Pierwsza funkcja zwróci obiekt określonej klasy, a druga funkcja o nazwie `print.klasa()` będzie odpowiedzialna za wyświetlenie tego obiektu.

W ten sposób działa większość funkcji statystycznych w programie R. Przykładowo wynikiem funkcji `summary()` jest obiekt klasy `summary`. Sama funkcja `summary()` nic nie wyświetla, ale jeżeli wynik tej funkcji nie zostanie nigdzie przypisany, to automatycznie wywoływana jest funkcja `print.summary()` (przeciążona wersja funkcji `print()`) odpowiedzialna za wyświetlenie podsumowania. Podobnie działają funkcje do rysowania wykresów z pakietów `lattice` i `ggplot2`. Więcej informacji o wykorzystywanym tu mechanizmie przeciążania znaleźć można w podrozdziale 1.6.2.3.

W pewnych sytuacjach możemy sobie nie życzyć, by wynik wyrażenia lub funkcji był wypisywany na konsoli przez funkcję `print()` (co jak wspominaliśmy dzieje

się automatycznie, jeżeli wynik nie jest do czegoś przypisany). Można temu zapobiec korzystając z funkcji `invisible()`. Działanie tej funkcji polega na tymczasowemu zapobiegnięciu wyświetlania jej argumentu w sytuacji, gdy nie będzie on do niczego przypisany.

Prześledźmy poniższy przykład. Definiujemy i wywołujemy dwie funkcje, obie są tożsamościami. Wyniki obu funkcji przypisujemy do zmiennych.

```
I1 <- function(x) x
I2 <- function(x) invisible(x)
```

Jak widzimy, w poniższym przypadku funkcja `invisible()` nie ma żadnego efektu. Dla obu funkcji obserwujemy takie samo zachowanie.

```
a <- I1(1)
a
## [1] 1
a <- I2(1)
a
## [1] 1
```

A teraz wyniku funkcji nie przypisujemy do zmiennej, powinien więc zostać wypisany na ekranie. Ale po użyciu funkcji `invisible()` wynik funkcji `I2()` nie jest wyświetlany.

```
I1(1)
## [1] 1
I2(1)
```

Funkcję `cat()` możemy również wykorzystać, aby zapisywać obiekty do pliku (zamiast wypisywać je na konsoli). Aby to zrobić należy argumentem `file` wskazać ścieżkę do pliku, do którego zapisane mają być obiekty. Więcej o zapisywaniu do plików przeczytać można w podrozdziale 1.6.6.

Wyświetlanie niesformatowane oznacza, że na ekranie wyświetlane są bezpośrednio napisy podane jako argumenty funkcji cat(), bez żadnych dodatków, takich jak [1] czy wcięcia.

Podsumowując, funkcja `cat()` służy do wyświetlania niesformatowanego, funkcja `print()` służy do wyświetlania sformatowanego. Funkcję `print()` można dowolnie przeciążać (czyli możemy sami określać jak wyświetlane mają być obiekty różnych klas). Funkcja `cat()` domyślnie nie jest przeciążona.

Do operacji na napisach w celu ich odpowiedniego wyświetlenia wykorzystuje się również funkcje `paste()` i `format()`. Funkcja `paste()` służy do łączenia wektorów napisów. Argumenty `sep=" "` i `collapse=NULL` określają separatory, którymi sklejane są wektory wejściowe.

W poniższym przykładzie sklejamy trzy argumenty w jeden łańcuch znaków.

```
paste("Ala", "ma", 5)
## [1] "Ala ma 5"
```

Parametrem sep, możemy określać, co ma separować kolejne argumenty.

```
paste("Ala", "ma", 5, sep="; ")
## [1] "Ala; ma; 5"
```

Jeżeli argumenty mają różną długość, to zadziała *recycling rule* (omówimy ją później).

```
paste("Jeszcze", 3:0, "...")
## [1] "Jeszcze 3 ..." "Jeszcze 2 ..." "Jeszcze 1 ..." "Jeszcze 0 ..."
```

Wynikiem sklejenia dwóch wektorów będzie wektor.

```
paste(1:5, letters[1:5], sep=" * ")
## [1] "1 * a" "2 * b" "3 * c" "4 * d" "5 * e"
```

Chyba, że określimy argument collapse, co spowoduje, że elementy tego wektora zostaną połączone.

```
paste(1:5, letters[1:5], sep=",", collapse="; ")
## [1] "1,a; 2,b; 3,c; 4,d; 5,e"
```

Funkcja `format()` służy do konwersji danego obiektu na typ znakowy zgodnie z ustalonym formatowaniem. Przy konwersji można określić, ile pól po kropce ma być wypisywanych, czy tekst ma być justowany do lewej czy do prawej, czy ma być wykorzystana notacja naukowa (z suffixem e+) itp. Poniżej kilka przykładów użycia funkcji `format()`.

```
format(11/3)
## [1] "3.666667"
```

Wyświetlmy tę liczbę w notacji naukowej.

```
format(11/3, sci = TRUE)
## [1] "3.666667e+00"
```

Wyświetlmy tę liczbę z maksymalnie dwoma cyframi znaczącymi.

```
format(11/3, digits = 2)
## [1] "3.7"
```

Wyświetlmy te liczby z dwoma miejscami po kropce dziesiętnej.

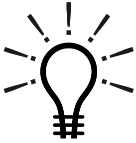
```
format(c(12,21)/3, nsmall = 2)
## [1] "4.00" "7.00"
format(c(12,21)/3, digits = 2, nsmall = 1)
## [1] "4.0" "7.0"
```

Funkcji do formatowania wyników jest znacznie więcej, np. funkcja `sprintf()` ma podobny sposób formatowania do funkcji o tej samej nazwie w języku C czy C++. Do konwersji na napis można użyć funkcji `toString()` i `encodeString()` czy, szczególnie użytecznej gdy chcemy zmienić kodowanie np. polskich znaków, `iconv()`.

Poniżej przedstawimy przykład dotyczący funkcji `cwhstring::formatFix()`, konwertującej wektor liczb na napisy o formacie określonym przez pozostałe argumenty.

```
require(cwhstring)
formatFix(c(pi, exp(1), 1, 1/pi),after=3, before=3)
## [1] "  3.142" "  2.718" "  1.000" "  0.318"
```

Argument `after` funkcji `formatFix()` określa liczbę miejsc po kropce, które mają być przedstawione, a argument `before` określa liczbę istotnych pozycji przed kropką. W wynikowym wektorze wszystkie napisy mają taką samą liczbę znaków.



Funkcja `require()` ładuje bibliotekę, podobnie jak funkcja `library()`. Różnica pojawia się w sytuacji, gdy danej biblioteki nie ma lub są z nią problemy. Domyślnie funkcja `library()` sygnalizuje błąd, co przerywa wykonywanie podprogramu, podczas gdy funkcja `require()` sygnalizuje ostrzeżenie, ale pozwala na kontynuację wykonywania podprogramu. Wynikiem funkcji `require()` w sytuacji, gdy żądana biblioteka nie jest dostępna jest wartość `FALSE`. Używając tej funkcji, programista może zaplanować awaryjne rozwiązanie, na wypadek braku danego pakietu.
Oczywiście, próba wywołania funkcji z pakietu, który nie został załadowany, zakończy się błędem.

## 1.6 Przyspieszamy

Po lekturze tego rozdziału czytelnik będzie już całkowicie gotów do korzystania z programu R. W kolejnych podrozdziałach przedstawimy składnię języka R, funkcje do operacji na danych, wprowadzenie do grafiki oraz podstawowe statystyki opisowe. Zakładamy, że czytelnik opanował materiał przedstawiony w poprzednich podrozdziałach.

Can one be a good data analyst without being a half-good programmer? The short answer to that is, 'No.' The long answer to that is, 'No.'

Frank Harrell
fortune(52)

### 1.6.1 Instrukcje warunkowe i pętle

Wiemy już jak pisać proste programy, w których instrukcje wykonywane są jedna po drugiej. Korzystając z instrukcji warunkowych i pętli możemy sterować przepływem wykonywania programu.

Poniżej przedstawione są instrukcje warunkowe, czyli polecenia pozwalające na wykonanie określonej listy instrukcji w zależności od tego czy określony warunek jest prawdziwy czy nie. Następnie przedstawione są pętle, czyli polecenia pozwalające na powtórzenie określonej listy instrukcji wielokrotnie.

#### 1.6.1.1 Instrukcja warunkowa if... else...

W języku R, tak jak w większości języków programowania, mamy możliwość korzystania z instrukcji `if... else...`. Umożliwia ona warunkowe wykonanie fragmentu kodu w zależności od prawdziwości pewnego warunku logicznego. Składnia instrukcji `if... else...` bez bloku `else` jest następująca

```
if (warunek_logiczny) {
        blok_instrukcji_1
}
```

a z blokiem `else`

```
if (warunek_logiczny) {
        blok_instrukcji_1
} else {
        blok_instrukcji_2
}
```

Jeżeli wartość `warunek_logiczny` jest prawdziwy (wartość logiczna równa `TRUE` lub liczbowa różna od 0), to wykonany zostanie `blok_instrukcji_1`. W przeciwnym przypadku wykonany zostanie `blok_instrukcji_2` (o ile zastosowano wariant z `else`). Jeżeli `blok_instrukcji_1` lub `blok_instrukcji_2` składa się tylko z jednej instrukcji, to można pominąć nawiasy klamrowe`{}`.

Zobaczmy, jak to wygląda na poniższym przykładzie. Zapis: `liczba %% 2 == 0` oznacza sprawdzenie, czy reszta z dzielenia przez 2 ma wartość 0, jezeli tak jest, to to wyrażenie przyjmuje wartość `TRUE`, w przeciwnym razie wartość `FALSE`.

```
liczba <- 1313
if (liczba %% 2 == 0) {
   cat("ta liczba jest parzysta\n")
} else {
   cat("ta liczba jest nieparzysta\n")
}
## ta liczba jest nieparzysta
```

Autor spodziewa się, że czytelnik nie wierzy na słowo, tylko sprawdzi jakim komunikatem zakończy się wpisanie tych poleceń.

Należy uważać, by słowo kluczowe `else` nie rozpoczynało nowej linii. Błędem zakończy się następujący ciąg poleceń:

```
if (1 == 0)
          cat("to nie moze byc prawda")
  else
          cat("wszystko ok")
```

Ale uwaga! Jeżeli te instrukcje znalazłyby się w ciele funkcji, to błąd nie zostałby zgłoszony. Wszystko by działało!

Dlaczego? Jak pamiętamy R to język interpretowany. Po zakończeniu drugiej linii tego przykładu interpreter nie spodziewa się kolejnych instrukcji, dlatego wykona (w jego mniemaniu już kompletną) instrukcję warunkową. Przechodząc do trzeciej linii o instrukcji `if` już nie pamięta, dlatego zostanie zgłoszony błąd składni. Poprawne użycie instrukcji `if` z wariantem `else` jest następujące (oba poniższe przykłady zadziałają poprawnie).

```
if (1 == 0) {
          cat("to nie moze byc prawda")
        } else {
          cat("wszystko ok")
        }
```

Mniej czytelnie ale poprawnie.

```
if (1 == 0)  cat("to nie moze byc prawda") else
                 cat("wszystko ok")
```

#### 1.6.1.2 Funkcja ifelse()

Powyżej omówiona instrukcja warunkowa bierze pod uwagę wartość tylko jednego warunku logicznego. Funkcja `ifelse()` pozwala na wykonanie ciągu działań w zależności od wektora warunków logicznych. Schemat użycia jest następujący:

```
ifelse(warunek_logiczny, instrukcja_1, instrukcja_2)
```

Warunek `warunek_logiczny` może być jedną wartością logiczną lub wektorem wartości logicznych. W wyniku wykonania funkcji `ifelse()` zwrócony zostanie

wartość lub wektor. Wynik będzie miał wartości opisane przez `instrukcja_1` w pozycjach odpowiadających wartości `TRUE` wektora `warunek_logiczny` oraz wartości opisane przez `instrukcja_2` w pozycjach odpowiadających wartości `FALSE` wektora `warunek_logiczny`.

Prześledźmy wyniki poniższych przykładów. najpierw przykład w którym tylko pierwszy argument jest wektorem.

```
ifelse(1:7 < 4, "mniej", "wiecej")
## [1] "mniej" "mniej" "mniej" "wiecej" "wiecej" "wiecej" "wiecej"
```

Teraz wszystkie argumenty są wektorami.

```
ifelse(sin(1:5)>0, (1:5)^2, (1:5)^3)
## [1]   1   4   9  64 125
```

A teraz wszystkie argumenty to pojedyncze wartości.

```
ifelse(1==2, " cos jest nie tak", "uff")
## [1] "uff"
```

### 1.6.1.3 Funkcja switch()

W przypadku omówionych powyżej instrukcji warunkowych, mieliśmy do czynienia z warunkiem logicznym, który mógł być prawdziwy lub fałszywy. Jednak w pewnych sytuacjach zbiór możliwych akcji, które chcemy wykonać jest większy. W takich sytuacjach sprawdza się instrukcja warunkowa `switch()` o następującej składni:

```
switch(klucz, wartosc1 = akcja1, wartosc2 = akcja2, ...)
```

Pierwszy argument powinien być typu znakowego lub typu wyliczeniowego `factor`. W zależności od wartości tego argumentu jako wynik zostanie zwrócona wartość otrzymana w wyniku wykonania odpowiedniej akcji.

W poniższym przykładzie sprawdzamy jaka jest klasa danej zmiennej i w zależności od tego wykonujemy jedną z wielu możliwych akcji. W bloku `klucz` sprawdzana jest klasa zmiennej `liczba`. Wiemy, że ta klasa to `'numeric'`, ponieważ zmienna `liczba` przechowuje wartość 1313. Dlatego też w wyniku wykona się wyłącznie druga akcja.

```
liczba <- 1313
switch(class(liczba),
   logical = ,
   numeric = cat("typ liczbowy lub logiczny"),
   factor = cat("typ czynnikowy"),
   cat("trudno okreslic")
)
## typ liczbowy lub logiczny
```

Jeżeli wartość klucza (pierwszego argumentu funkcji `switch()`) nie pasuje do etykiety żadnego z kolejnych argumentów, to wynikiem instrukcji `switch()` jest wartość argumentu nie nazwanego (czyli domyślną akcją w powyższym przykładzie jest wyświetlenie napisu `"trudno okreslic"`). Jeżeli wartość klucza zostanie

dopasowana do etykiety jednego z kolejnych argumentów, ale nie jest podana żadna związana z nim akcja, to wykonana zostanie akcja dla kolejnego argumentu. Innymi słowy, jeżeli klucz miałby wartość `"logical"`, to ponieważ nie jest wskazana wartość dla argumentu `logical=` zostanie wykonana akcja wskazana przy kolejnej z etykiet, czyli `"numeric"`.

Argumenty funkcji `switch()` korzystają z mechanizmu leniwego wartościowania (ang. *lazy evaluation*, więcej informacji nt. tego mechanizmu przedstawimy w podrozdziale 1.6.3). W językach z gorliwą ewaluacją, takich jak Java czy C++ przekazywanie argumentów funkcji polega na wyznaczeniu wartości kolejnych argumentów i na przekazaniu do funkcji wyłącznie wyznaczonych wartości. Gdyby tak było w powyższym przykładzie, to przed wywołaniem funkcji `switch()` wyznaczone byłyby wartości wszystkich argumentów, czyli wykonane byłyby wszystkie funkcje `cat()`. Tak się jednak nie dzieje, ponieważ w programie R argumenty przekazywane są w sposób leniwy i funkcja `switch()` sama decyduje, którą z instrukcji wykonać (wartość którego z argumentów wyznaczyć).

#### 1.6.1.4 Pętla for

Najpopularniejszą pętlą w większości języków programowania jest pętla `for`. Jest ona oczywiście również dostępna w języku R. Poniżej przedstawiamy jej szablon.

```
for (iterator in kolekcja) {
        blok_instrukcji
}
```

Przy użyciu tej pętli `blok_instrukcji` będzie wykonany tyle razy ile elementów znajduje się w obiekcie `kolekcja` (może być to wektor lub lista). Zmienna `iterator` w każdym okrążeniu pętli przyjmować będzie kolejną wartość z wektora lub listy `kolekcja`. Jeżeli `blok_instrukcji` składa się tylko z jednej instrukcji, to możemy pominąć nawiasy klamrowe.

Zacznijmy od bardzo typowego przykładu, w którym `kolekcja` to wektor liczb. Poniższa pętla wykona się dla każdego elementu wektora `1:5`.

```
for (i in 1:5) {
   cat(paste("aktualna wartosc zmiennej i to ", i, "\n"))
}
## aktualna wartosc zmiennej i to 1
## aktualna wartosc zmiennej i to 2
## aktualna wartosc zmiennej i to 3
## aktualna wartosc zmiennej i to 4
## aktualna wartosc zmiennej i to 5
```

Elementy wektora lub listy `kolekcja` mogą być dowolnego typu. W przypadku listy, te elementy mogą być wręcz różnych typów. Elementy wektora lub listy `kolekcja` nie muszą być różne, mogą się powtarzać. Poniżej przedstawiamy przykład, w którym `iterator` przebiega po wartościach wektora napisów. Poniższa pętla wykona się dla każdego elementu wektora `indeksy`.

```
indeksy <- c("jablka","gruszki","truskawki","gruszki")
for (i in indeksy) {
    cat(paste(i, "\n"))
}
## jablka
## gruszki
## truskawki
## gruszki
```

Jeżeli wewnątrz pętli wystąpi błąd, przerywa on wykonywanie pętli. Aby sprawdzić w którym kroku wystąpił błąd należy sprawdzić wartość zmiennej `iterator`.

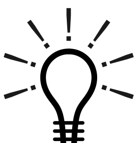
Bardzo często za wektor `kolekcja` podaje się wektor kolejnych liczb całkowitych. Umożliwia to zapisywanie wyników z kolejnych okrążeń pętli w wektorze lub w macierzy. W takich sytuacjach do wyznaczania wektora indeksów pętli wygodnie jest się posłużyć funkcją `seq_along()`. Argumentem tej funkcji jest wektor dowolnych wartości, a wynikiem jest wektor kolejnych liczb naturalnych od 1 do długości wektora będącego argumentem. Poniższy kod będzie miał identyczny wynik jak przykład z owocami powyżej, ale pętla indeksowana jest liczbami naturalnymi. Posługując się liczbowym indeksem `i` możemy wyniki zapisywać do macierzy lub innego obiektu, którego nie można indeksować nazwami owoców.

Funkcja seq_along(x) jest bezpieczniejsza niż sekwencja 1:length(x), ponieważ dla pustego x ta pierwsza zwraca pusty wektor a ta druga wektor 1:0! Co bywa przyczyną trudnych do wyśledzenia błędów.

Poniższa pętla wykona się dla każdego elementu wektora `indeksy` ale zmienna `i` będzie przyjmowała wartości od 1 do 4.

```
for (i in seq_along(indeksy)) {
    cat(paste("Element", i, "to", indeksy[i], "\n"))
}
## Element 1 to jablka
## Element 2 to gruszki
## Element 3 to truskawki
## Element 4 to pomarancze
```

Zazwyczaj podobny efekt do pętli `for` możemy uzyskać stosując funkcje z rodziny `*apply()` (np. `lapply()`). Funkcje te będą przedstawione w podrozdziale 2.1.4. Żadne z tych rozwiązań nie jest absolutnie lepsze, można więc korzystać z dowolnego z nich w zależności od tego, które jest łatwiej w konkretnej sytuacji zastosować (zapisać). Najczęściej jednak używając funkcji z rodziny `*apply()` otrzymuje się bardziej elegancki zapis, w wielu przypadkach też wynik wyznaczony będzie szybciej ponieważ R jest optymalizowany do pracy na wektorach.



W przykładzie powyżej używaliśmy funkcji `seq_along(x)` by wyznaczyć sekwencje indeksów od 1 do długości wektora. Alternatywny zapis `1:length(x)` źle zadziała jeżeli argument *x* ma zerową długość, lepiej więc używać funkcji `seq_along()`.

```
x <- c(2,1,5)
seq_along(x)
## [1] 1 2 3
1:length(x)
## [1] 1 2 3
```

Przykład z wektorem o zerowej długości. Niepoprawne obsługiwanie pustych wektorów jest częstą przyczyną trudnych w wyśledzeniu błędów.

```
x <- NULL
seq_along(x)
## integer(0)
1:length(x)
## [1] 1 0
```

Poniżej przedstawiamy interesujący przykład użycia pętli for. Iterator `obiekt` przesuwa się po wektorze zwróconym przez funkcję `ls()`, czyli wektorze obiektów w obecnej przestrzeni nazw. Przy każdym kroku pętli wywoływana jest instrukcja `cat()` wypisująca na ekranie nazwę obiektu oraz rozmiar obiektu w bajtach (wynik funkcji `object.size()`, która jako argument przyjmuje obiekt do zmierzenia a nie nazwę, dlatego potrzebne jest jeszcze wywołanie funkcji `get()`).

```
for (obiekt in ls())
     cat("zmienna:",obiekt,"rozmiar:",object.size(get(obiekt)),"\n")
## zmienna: macierz rozmiar: 80200
## zmienna: obiekt rozmiar: 96
## zmienna: tmp rozmiar: 544
## zmienna: vec1 rozmiar: 80040
```

#### 1.6.1.5 Pętla while

Pętla `for` ma z góry określoną liczbę „obrotów". Gdy nie wiemy ile powtórzeń pętli jest wymaganych, aby uzyskać zamierzony efekt, możemy wykorzystać pętlę `while` o szablonie przedstawionym poniżej.

```
while (warunek_logiczny) {
        blok_instrukcji
}
```

W tej pętli `blok_instrukcji` będzie wykonywany tak długo, jak długo prawdziwy jest `warunek_logiczny`. Oczywiście, należy zadbać o to, by taka sytuacja kiedykolwiek zaistniała, a więc by pętla kiedyś się zakończyła.

W poniższym przykładzie pętla będzie się wykonywać póki warunek `i < 3` będzie prawdziwy.

```
i <- 0
while(i < 3) {
   cat(paste("juz", i, "\n"))
   i <- i+1
}
## juz 0
## juz 1
## juz 2
```

Instrukcja `while()` może być bardzo przydatna, jeżeli mamy do czynienia z losowością, i chcemy by zdarzył się pewien określony warunek. W takim przypadku wystarczy powtarzać losowanie tak długo aż ten warunek się nie wydarzy.

W przykładzie poniżej powtarzamy losowanie 10 liczb z przedziału 1 do 100 (to robi instrukcja `sample(100,10)`) tak długo, aż nie wylosuje się kombinacja w któ-

rej występują dwie liczby różniące się o nie więcej niż 1. Warto zwrócić uwagę, że `blok_instrukcji` w tym przypadku jest pusty, a `warunek_logiczny` został tak rozbudowany, by wylosować w nim liczby, zapisać do pomocniczego wektora `x` i sprawdzić czy ten wektor spełnia pożądane warunki. Wyrażenie `warunek_logiczny` jest typową cebulką, w której wynik wywołania jednej funkcji przekazywany jest do kolejnej, rozszyfrowując co taka cebulka robi, warto zacząć od środka.

Czytając tę cebulkę od środka: losujemy 10 liczb ze zbioru 1..100, porządkujemy je rosnąco, zapisujemy ten wektor do zmiennej x, liczymy różnice pomiędzy sąsiednimi wyrazami wektora, wyznaczamy najmniejszą z różnic pomiędzy kolejnymi wyrazami, sprawdzamy czy jest ona większa od 1. Jeżeli jest większa od 1 to powtarzamy zabawę.

```
while( min(diff(x <- sort(sample(100,10)))) > 1 ) {}
```

Zobaczmy co się wylosowało, że pętla została zakończona.

```
x
##  [1] 10 15 19 20 39 61 68 71 88 91
```

#### 1.6.1.6 Pętla repeat

Innym rodzajem pętli jest pętla `repeat` o składni przedstawionej poniżej.

```
repeat {
    blok_instrukcji
}
```

Działanie tej pętli polega na powtarzaniu `blok_instrukcji` tak długo aż .... Właśnie, nie ma tu żadnego warunku stop! Działanie zostanie przerwane wyłącznie w wyniku wygenerowania błędu lub użycia instrukcji `break` (ta instrukcja przerywa wykonywanie wszystkich rodzajów pętli, również pętli `for` i `while`).

Moim zdaniem, korzystanie z tego sposobu kończenia pętli jest w bardzo złym stylu.

Przyjrzymy się poniższemu przykładowi.

```
repeat {
   cat("uda sie czy nie?\n")
   if (runif(1) < 0.1)
              break
}
## uda sie czy nie?
## uda sie czy nie?
```

Instrukcję `break` można wykorzystywać w każdym rodzaju pętli, podobnie w każdej pętli można wykorzystywać instrukcję `next`. Pierwsza przerywa działanie pętli, druga powoduje przerwanie wykonywania aktualnej iteracji pętli oraz przejście do kolejnej iteracji.

### 1.6.2 Funkcje

Często pisząc większe skrypty pewne fragmenty kodu powtarzają się wielokrotnie i/lub są używane w różnych miejscach. Czasem należy wykonać pewien schemat instrukcji, być może z niewielką różnicą w argumentach. W takich sytuacjach wygodnie jest napisać funkcję, która uprości program i zwiększy jego czytelność. Generalnie rzecz biorąc zamykanie kodu w funkcjach jest dobrym zwyczajem, zwiększa to czytelność i zazwyczaj skraca kod, a tym samym wpływa na zmniejszenie liczby błędów, które zawsze gdzieś w kodzie się znajdą. Jest wiele wskazówek jak pisać dobre funkcje, zarówno elastyczne, reużywalne, robiące jedną rzecz, a więc łątwe w opisie i zapamiętaniu czemu dana funkcja służy. Ponieważ jednak nie jest to

podręcznik do programowania poprzestanę na dwóch uwagach. Funkcje powinny być tak tworzone, by nie korzystały ze zmiennych globalnych (parametry do działania powinny być przekazane poprzez argumenty, używanie zmiennych globalnych najczęściej prowadzi do trudnych w wykryciu błędów), dzięki temu stają się izolowane i łatwiej je wykorzystywać w różnych kontekstach, różnych programach. Funkcje powinny być możliwie krótkie, ułatwi to ich modyfikacje i śledzenie poprawności, łatwiej też zrozumieć co dana funkcja robi. Jeżeli jakaś funkcja znacznie się rozrosła, to z pewnością można i warto podzielić ją na mniejsze funkcje. Poniżej przedstawiamy schemat deklaracji funkcji.

```
function(lista_argumentow) {
   blok_instrukcji
}
```

Jeżeli `blok_instrukcji` składa się z jednej instrukcji, to można pominąć nawiasy klamrowe. Zawartość `lista_argumentow` to lista par (być może pusta): nazwa argumentu, jego domyślna wartość (opcjonalnie). Funkcje w języku R są traktowane jak zwykłe obiekty. Konsekwencje takiego rozwiązania zostaną szczegółowo przedstawione później.

Przyjrzymy się takiemu przykładowi. Definiujemy funkcję i przypisujemy ją do zmiennej `funkcja1`. Następnie ją wywołujemy.

```
funkcja1 <- function() {
   cat("Dzisiaj jest ")
   cat(format(Sys.time(), "%A %B %d"))
}
funkcja1()
## Dzisiaj jest wtorek listopad 05
```

Przypisujemy do zmiennej `funkcja2` wartość zmiennej `funkcja1`. Możemy ją wywołać tak samo, jak funkcję `funkcja1`.

```
funkcja2 <- funkcja1
funkcja2()
## Dzisiaj jest wtorek listopad 05
```

Na początku tego przykładu zdefiniowana została funkcja, która następnie została przypisana do zmiennej `funkcja1`. Ta zmienna jest teraz zmienną typu funkcyjnego. Aby edytować ciało zmiennej możemy posłużyć się funkcją `edit()` lub `fix()`. Poniższa instrukcja otworzy ciało funkcji w oknie edytora.

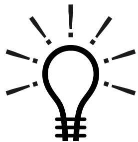
```
edit(funkcja1)
```

Z punktu widzenia R, funkcja jest takim samym obiektem, jak każdy inny obiekt. Nazwa funkcji nie jest związana z jej definicją, a jedynie z nazwą zmiennej, w której ta funkcja jest zapamiętana.

Jeżeli chcemy, by do definiowanych funkcji można było przekazywać argumenty, to w deklaracji funkcji należy umieścić nazwy tych argumentów w wektorze `lista_argumentow`, rozdzielając kolejne argumenty przecinkami.

Funkcje mogą przekazywać (potocznie mówiąc zwracać) wartości. Za wynik funkcji przyjmowana jest wartość wyznaczona w ostatniej linii ciała funkcji. Innym sposobem przekazywania wartości jest wykorzystanie instrukcji `return()`. Powoduje ona przerwanie wykonywania funkcji oraz przekazanie jako wyniku wartości będącej argumentem polecenia `return()`.
Definiujemy nową funkcję. Wykorzystaną tutaj funkcję `sort()` opiszemy w innym miejscu, ale jak można odgadnąć sortuje ona elementy wektora.

```
wyswietl3Najmniejsze <- function(wektor) {
   posortowane <- sort(wektor)
   posortowane[1:3]
}
lLiczby <- c(11, 3, 10, 1, 0, 8)
(wynik <- wyswietl3Najmniejsze(lLiczby))
## [1] 0 1 3
```



Nazwy argumentów funkcji potrafią być długie, jednak nie trzeba ich całych podawać! Zamiast pełnej nazwy argumentu wystarczy podać fragment nazwy taki, który jednoznacznie identyfikuje argument.
Podobny mechanizm funkcjonuje, gdy argument może przyjąć wartość z pewnego zbioru wartości. W tym przypadku nie trzeba wskazywać pełnej nazwy wybranej wartości, wystarczy taki fragment, który jednoznacznie określa o którą wartość chodzi. Za takie częściowe dopasowanie odpowiedzialna jest funkcja `match.arg()`. Przykład mechanizmu skrótów przedstawiony jest poniżej.

Poniżej deklarujemy funkcję z dwoma argumentami. Wywołujemy ją jawnie wskazując wartości obu argumentów, albo używając skrótów nazw.

```
funkcja <- function(liczba = 5, poziom = "sredni")
             cat(paste(liczba, poziom,"\n"))
funkcja(3, "duzy")
## 3 duzy
```

Wystarczy podać kilka pierwszych liter argumentu, o ile można jednoznacznie określić o który argument chodzi.

```
funkcja(po = "duzy")
## 5 duzy
funkcja(p = "maly", li = 1313)
## 1313 maly
```

#### 1.6.2.1 Argumenty domyślne

Definiując funkcję możemy określić domyślne wartości dla kolejnych jej argumentów. Jeżeli to uczynimy, to gdy przy wywołaniu funkcji dany argument nie będzie jawnie podany, wykorzystana będzie jego domyślna wartość.

Wywołując funkcję, wartości jej argumentów możemy podawać w dowolnej kolejności. Jeżeli jednak zrezygnujemy z kolejności określonej w liście argumentów, to poprzez nazwę musimy wskazać, który argument wprowadzamy. Gdy argumentów domyślnych jest więcej, możemy w wywołaniu funkcji pomijać te, któ-

rych modyfikować nie chcemy, a w liście argumentów zamiast wartości zostawić puste miejsce.

Wszystkie wymienione możliwości zostały przedstawione na przykładzie poniżej. Deklarujemy funkcje z trzema argumentami. Możemy wywołać tę funkcję z dowolną kombinacją i kolejnością argumentów, poniżej określamy tylko pierwszy argument, pozostałe będą domyślne.

```
wyswietlNajmniejsze <- function(wektor, do = 3, od = 1) {
   posortowane <- sort(wektor)
   posortowane[od:do]
}
wyswietlNajmniejsze(lLiczby)
## [1] 0 1 3
```

Określamy dwa pierwsze argumenty.

```
wyswietlNajmniejsze(lLiczby, 5)
## [1] 0  1  3  8  10
```

Określamy dwa argumenty, pierwszy i trzeci (musimy go wskazać przez nazwę).

```
wyswietlNajmniejsze(lLiczby, od = 5)
## [1] 10  8  3
```

Kolejność argumentów może być dowolna.

```
wyswietlNajmniejsze(do = 5, wektor = lLiczby)
## [1] 0  1  3  8  10
```

Domyślne argumenty możemy pomijać zostawiając puste miejsce w liście argumentów.

```
wyswietlNajmniejsze(losoweLiczby, , 3)
## [1] 3
```

W deklaracji funkcji nie trzeba specyfikować nazw jej wszystkich możliwych argumentów, o ile nie są one wykorzystywane w danej funkcji. Jeżeli chcemy pozostawić możliwość podawania dodatkowych argumentów do funkcji, to w liście argumentów można umieścić ... (wielokropek). Te nadmiarowe, nie wymienione z nazwy argumenty, nie będą wykorzystane w ciele tej funkcji, ale mogą być przekazane dalej w wewnętrznych wywołaniach kolejnych funkcji.

Poniżej przedstawiamy przykład wywołania funkcji z nadmiarowymi argumentami, które koniec końców zostaną przekazane do funkcji `plot()`. Wywołujemy funkcję `narysujNajmniejsze()` z dodatkowymi argumentami (`lwd`, `type` i `col`) nie wymienionymi jawnie w liście argumentów. Funkcja `list()` pozwala na wyłuskanie obiektów z argumentu `...`.

```
narysujNajmniejsze <- function(wektor, ile = 3, ...) {
   posortowane <- sort(wektor)
   plot(posortowane[1:ile], ...)
}
narysujNajmniejsze(lLiczby, ile=20, lwd=3, type="l", col="black")
```

I jeszcze jeden przykład na przekazywanie argumentów.

```
funkcjaPrzekaznik <- function(...) {
    print(list(...))
  }
funkcjaPrzekaznik(arg1=10, arg2=1313)
## $arg1
## [1] 10
##
## $arg2
## [1] 1313
```

Przekazywanie argumentów w ten sposób jest bardzo wygodne przy pisaniu wrapperów. Nie trzeba wtedy jawnie podawać (być może bardzo długiej) listy wszystkich argumentów opakowywanej funkcji.

Wrapper to funkcja opakowująca inną, standardową funkcję. Wrappery są wykorzystywane najczęściej po to, aby ujednolicić sposób wywoływania interesujących nas funkcji.

Nie tylko twórca funkcji może wskazywać wartości domyślne dla argumentów tej funkcji. Może to zrobić też użytkownik, określając jakie wartości mają być uznawane za domyślne dla danego argumentu. Można to zrobić korzystając z funkcji `options()`, pozwalającej na globalne określanie domyślnych wartości dla pewnych argumentów lub korzystając z pakietu `Defaults` umożliwiającego ustawianie domyślnej wartości dla argumentów wskazanej funkcji.

#### 1.6.2.2 Funkcje anonimowe

W pewnych sytuacjach wygodnie jest jako argument funkcji podać funkcję. Można to zrobić na różne sposoby. Jednym z nich jest zdefiniowanie takiej funkcji wcześniej, przypisanie jej do zmiennej i podanie jako argument danej zmiennej. Ponieważ w tym przypadku nazwa zmiennej nie ma żadnego znaczenia, dlatego nie musimy jej podawać. Wygodniej jest podać ciało funkcji bezpośrednio jako argument. Mówimy w tym przypadku o funkcji anonimowej, ponieważ nie jest ona przypisana do żadnej zmiennej, przez co też nie ma nazwy.

W pewnym sensie wszystkie funkcje są anonimowe, to zmienne mają nazwy a nie przypisane do nich funkcje.

Tak, można funkcję podawać jako argument do innej funkcji.

Wykorzystanie funkcji anonimowych obrazuje poniższy przykład. Korzystamy tu z polecenia `sapply()`, które wykonuje zadaną funkcję (drugi argument funkcji `sapply()`) dla każdego elementu wektora lub listy (pierwszy argument polecenia `sapply()`).

```
pewnaFunkcja <- function(x) {
   x^2 + 3
}
sapply(c(1,2,3), pewnaFunkcja)
## [1]  4  7 12
```

Wywołanie funkcji sapply z użyciem funkcji anonimowych.

```
sapply(c(1,2,3), function(x) x^2+3)
##  [1]   4   7  12
##
```

Przekazywanie funkcji jako argumentu może wyglądać egzotycznie, ale jak przekonamy się w kolejnych podrozdziałach jest to wyjątkowo przydatny mechanizm, dający wiele możliwości.

#### 1.6.2.3 Funkcje polimorficzne

Jeżeli ktoś nie lubi słowa polimorficzność, to niech lepiej nie czyta tego podrozdziału.

Kolejnym mechanizmem obecnym w języku R, przypominającym cechy języka obiektowego, jest możliwość definiowania funkcji polimorficznych. Mechanizm ten, nazywany jest też przeciążaniem funkcji.

Przypuśćmy, że chcemy przeciążyć funkcję `plot()`, a więc sprawić, aby dla określonych argumentów, funkcja ta zachowywała się w specyficzny, określony sposób (nie każdą funkcję można przeciążyć, ale do tego tematu wrócimy w podrozdziale 2.1.8). W tym celu należy zdefiniować nową funkcje i przypisać ją do zmiennej o nazwie `plot.typ`, gdzie `typ` to nazwa klasy, dla której ta funkcja jest dedykowana (np. `factor`). Wywołanie funkcji o nazwie bez suffiksu `.typ` ale z argumentem klasy `typ` spowoduje wywołanie funkcji z suffiksem `.typ`. Jeżeli więc funkcja`plot()` jest polimorficzna, a obiekt `wynik` jest klasy `summary` to otrzymamy ten sam efekt pisząc `plot(wynik)` i `plot.summary(wynik)`.

Przykładowo, funkcja `plot()` zazwyczaj rysuje coś na ekranie. Możemy jednak zażyczyć sobie, aby w zależności od klasy pierwszego argumentu zachowywała się inaczej. Poniżej przedstawiamy przykład, w którym funkcja `plot()` została tak przeciążona, by dla argumentów o typie logicznym zamiast rysować wypisywała wartość argumentu.

Poniżej deklarujemy funkcję o nazwie `plot.logical`. Funkcja ta będzie wywołana, jeżeli za nazwę funkcji podamy `plot` a argument będzie typu `logical`.

```
plot.logical <- function(obj) {
  cat(ifelse(obj,"prawda","nieprawda"))
}
plot(1:10 < 3)
## prawda prawda nieprawda nieprawda nieprawda nieprawda nieprawda
    nieprawda nieprawda nieprawda
```

Pamiętamy, że korzystając z funkcji `class()` możemy dowolnie modyfikować nazwę klasy danego obiektu. Dzięki tej możliwości i dzięki mechanizmowi przeciążania funkcji możemy tworzyć obiekty określonej klasy i przeciążać dla nich podstawowe funkcje. Jeżeli tak zrobimy, to inni użytkownicy będą mogli w intuicyjny sposób korzystać z naszych nowych funkcjonalności. W ten sposób działają funkcje `plot()` i `summary()` dla takich typów jak `lm`, `factor`, `formula` itp. Jeżeli argumentem funkcji `summary()` jest obiekt klasy `lm`, to uruchamiana jest funkcja `summary.lm()`, która prezentuje w tekstowej postaci poszczególne elementy modelu liniowego (do tego tematu wrócimy w części poświęconej statystyce). Innymi, często przeciążanymi funkcjami są `predict()`, `anova()`, `print()` oraz operatory.

Korzystając z funkcji `methods()` możemy sprawdzić, czy zadeklarowane są jakieś przeciążone wersje interesującej nas funkcji lub też, czy są przeciążone funkcje dla jakiejś interesującej nas klasy. Warto zobaczyć jaki jest wynik polecenia `methods("plot")`, aby przekonać się jak wiele jest przeciążeń funkcji `plot()`.

Wyświetlmy listę przeciążonych wersji funkcji `plot()`.

```
methods(plot)
##  [1] plot.acf*          plot.agnes*        plot.correspondence*
##  [4] plot.data.frame*   plot.Date*         plot.decomposed.ts*
##  [7] plot.default       plot.dendrogram*   plot.density
## [10] plot.diana*        plot.ecdf          plot.factor*
```

```
## [13] plot.formula*       plot.hclust*        plot.histogram*
## [16] plot.lm             plot.mca*           plot.medpolish*
## [19] plot.mlm            plot.mona*          plot.partition*
## [22] plot.POSIXct*       plot.ppr*           plot.prcomp*
## [25] plot.princomp*      plot.profile*       plot.table*
##
##    Non-visible functions are asterisked
```

Nie każda funkcja może być przeciążona. Aby program R wiedział, że jakaś funkcja jest generyczna (czyli może być przeciążana) należy oznaczyć ją używając funkcji `UseMethod()`. Kompletny opis działania tej funkcji (a zarazem technicznych aspektów działania funkcji przeciążanych) wykracza poza zakres tej książki, ograniczmy się więc jedynie do przykładu. W poniższym przykładzie funkcja `rozmiar()` wyznacza i przekazuje liczbę elementów w danym obiekcie, bez względu czy jest to wektor, macierz czy ramka danych.

W pierwszej linii definiujemy funkcję `rozmiar`, która będzie przeciążana.

```
rozmiar <- function(x) UseMethod("rozmiar")
```

Musimy teraz określić jej domyślne zachowanie. Funkcja `rozmiar.default()` zostanie uruchomiona jeżeli nie zostanie dopasowana inna wersja funkcji `rozmiar()`.

```
rozmiar.default    <- function(x) length(x)
```

Następnie określamy jej zachowanie dla konkretnych klas argumentów.

```
rozmiar.character  <- function(x) length(x)
rozmiar.matrix     <- function(x) dim(x)[1] * dim(x)[2]
rozmiar.array      <- function(x) prod(dim(x))
```

Wykonajmy teraz test, jak zadziała ta funkcja dla wektora liczb.

```
rozmiar(10:1)
## [1] 10
```

Wywołujemy funkcję `rozmiar()` dla macierzy.

```
rozmiar(matrix(0,10,10))
## [1] 100
```

#### 1.6.2.4 Funkcje a zasięg

Każdy z trzech operatorów `->`, `<-` i `=` przypisuje wartości do zmiennej o lokalnym zasięgu (w aktualnym środowisku używając terminologii R). A więc taki operator użyty w funkcji zmienia wartość zmiennej lokalnie w funkcji. W języku R są dostępne również dwa inne operatory przypisania, mianowicie `->>` i `<<-`. Ich działanie różni się tym, że przypisują wartość do zmiennej o zasięgu globalnym, a więc zmiany widoczne są poza funkcją.

W przykładzie poniżej definiujemy nową funkcję, w której będziemy przypisywać wartości do zmiennych lokalnych. Zainicjujmy wartość dwóch zmiennych. Pierwsze przypisanie to normalne, lokalne przypisanie. Drugie przypisanie to globalne przypisanie, modyfikowana jest zmienna `zmienna2` w zewnętrznej przestrzeni nazw.

```
zmienna1 <- 1
zmienna2 <- 1
przyklad1 <- function() {
        zmienna1 <- 2
        zmienna2 <<- 2
        cat(paste("zmienna1:", zmienna1, "zmienna2:", zmienna2, "\n"))
}
```

Stan zmiennych globalnych przed uruchomieniem funkcji.

```
cat(paste("zmienna1:", zmienna1, "zmienna2:", zmienna2, "\n"))
## zmienna1: 1 zmienna2: 1
```

Stan zmiennych lokalnych wewnątrz funkcji.

```
przyklad1()
## zmienna1: 2 zmienna2: 2
```

Stan zmiennych globalnych po uruchomieniu funkcji.

```
cat(paste("zmienna1:", zmienna1, "zmienna2:", zmienna2, "\n"))
## zmienna1: 1 zmienna2: 2
```

Operator = powinien być stosowany tylko w „najbardziej zewnętrznym” poziomie zagnieżdżenia (tam jest równoważny operatorom <- i ->). Jeżeli wywołujemy funkcję, to w specyfikacji jej argumentów operatory = i <- mają odmienne znaczenia!!! Operator „strzałkowy” oznacza przypisanie wartości, podczas gdy operator = wskazuje, który argument funkcji jest określany. Różnice te demonstruje poniższy przykład.

Ta instrukcja się nie wykona, zostanie zinterpretowana jako próba wskazania wartości argumentu o nazwie `liczby`, który nie jest argumentem funkcji `plot()`.

```
plot(liczby = 1:100)
## Error in plot(liczby = 1:100) : argument "x" is missing, with no
    default
```

Ta instrukcja wykona się poprawnie, operacja podstawienia przekazuje wynik i to on będzie narysowany na ekranie.

```
plot(liczby <- 1:100)
```

### 1.6.2.5 Funkcje a operatory

W programie R mamy możliwość definiowania własnych operatorów. Operatorem może być dowolny ciąg znaków otoczony znakami `%`. Z technicznego punktu widzenia operatory są zwykłymi funkcjami, różnią się jedynie metodą ich wywołania. Poniżej przykład zdefiniowania i użycia operatora `%v%`.

Poniżej definiujemy operator tak jak zwykłą funkcję dwuargumentową, a używamy w sposób typowy dla operatorów.

```
"%v%" <- function(x,y) x*y+x+y
1 %v% 2
## [1] 5
```

#### 1.6.2.6 Obiekty wywołań funkcji

Obiektem może być polecenie wykonania funkcji. Taki obiekt inicjuje się funkcją `call()`. Poniżej zmienna `wywolanie` przechowuje wywołanie funkcji `round()`.

```
wywolanie <- call("round", 10.5)
wywolanie
## round(10.5)
```

Funkcja `eval` interpretuje i wykonuje `wywolanie`.

```
eval(wywolanie)
## [1] 10
```

W tym przykładzie obiekt `cl` przechowuje informacje o poleceniu wykonania funkcji `round()`. Ta funkcja nie została jeszcze wykonana, jej wynik nie jest jeszcze określony. Polecenie wykonania można wykonać używając funkcji `eval()`. Zwróćmy uwagę, że nazwa funkcji będącej argumentem funkcji `call()` jest łańcuchem znaków, można więc nazwę funkcji do wykonania skonstruować tak jak konstruuje się napisy, np. składając je za pomocą funkcji `paste()`.

Ponieważ polecenie wykonania jest zwykłym obiektem klasy `call`, to może ono być przekazywane jako argument funkcji, zapisywane do plików itp. Efekt podobny do złożenia funkcji `call()` i `eval()` ma funkcja `do.call()`. Wywołuje ona określoną funkcję ze wskazanymi argumentami. Nazwę funkcji podaje się jako zwykły napis, może więc ją dynamicznie konstruować.

Przedstawmy przykład użycia funkcji `do.call`. Dynamicznie tworzymy nazwy funkcji do wywołania.

```
(nazwyFunkcji <- paste("r", c("unif", "norm", "exp"), sep=""))
## [1] "runif" "rnorm" "rexp"
```

Wywołujemy te funkcje przekazując do nich dodatkowe argumenty.

```
sapply(nazwyFunkcji, FUN=do.call, list(4))
##            runif      rnorm      rexp
##  [1,] 0.55210300  1.0712174 3.3269195
##  [2,] 0.34391913  0.3055000 0.6680466
##  [3,] 0.78100916 -1.1005831 0.6848731
##  [4,] 0.24982938  0.6977821 0.5723458
```

#### 1.6.2.7 Inne zagadnienia związane z funkcjami

„Oczywiście” możemy definiować funkcje zagnieżdżone. Oznacza to, że w środku jednej funkcji definiujemy inną funkcję, którą przypisujemy do lokalnej zmiennej (a więc zmiennej nie widocznej poza funkcją). Poniżej przykład dla zagnieżdżonych funkcji.

Funkcja to zwykły obiekt, możemy więc np. zapisać ją pliku, np. poleceniem `save()`!

```
zewnetrzna <- function(x) {
   wewnetrzna <- function(y) {
      print(y)
   }
  wewnetrzna(x)
}
```

W programie R jest tak wiele użytecznych funkcji, że nie sposób pokazać tutaj wszystkich. W tej książce przedstawiamy tylko ułamek z wielkiego zbioru wszystkich funkcji (jak duży to zbiór łatwo ocenić przeglądając liczbę wpisów w skorowidzu). Rozdział ten zamkniemy przedstawieniem jeszcze dwóch ciekawych funkcji, mianowicie: `.First()` i `.Last()`. Funkcje te są wywoływane odpowiednio na samym początku pracy oraz przed zamknięciem środowiska R. Możemy definiować własne wersje tych funkcji, personalizując tym samym środowisko R. Poniżej przykład zmiany definicji tych funkcji.

Co program R ma robić po uruchomieniu środowiska? A co ma robić przed zamknięciem środowiska?

```
.First <- function()
     options(prompt="# ", continue="- \t")
.Last <- function()
     cat("baj baj")
```

Jeżeli wprowadzimy te funkcje, a następnie zapiszemy całe środowisko w domyślnej przestrzeni nazw (jest on ładowany przy każdym uruchomieniu R), to przy kolejnym uruchomieniu programu R wykona się funkcja `.First()`. W powyższym przykładzie zmienia ona między innymi znak zachęty ze znaku > na znak #. Przy zamykaniu R wykona się funkcja `.Last()`, która wyświetli dwa słowa pożegnania.

You can be maximally lazy, but still be efficient.
Kevin Murphy (describing the implementation of an algorithm)
fortune(12)

### 1.6.3 Leniwa ewaluacja

O mechanizmie leniwej ewaluacji (ang. *lazy evaluation*) pisaliśmy przy okazji funkcji `switch()`. Ale mechanizm ten nie dotyczy tylko tej jednej funkcji, ale wszystkich funkcji w programie R. W przypadku każdej funkcji, wartości jej argumentów zdefiniowane w miejscu wywołania funkcji nie są wyznaczane, póki ich wartość nie jest potrzebna. Co więcej, wyrażenie będące argumentem funkcji nie musi mieć sensu w miejscu wywołania, może odnosić się do zmiennych, które w miejscu wywołania funkcji nie są zdefiniowane. Ważne jest by wartość argumentów dało się wyznaczyć w miejscy gdzie będą potrzebne. W większości niskopoziomowych języków programowania stosowana jest gorliwa ewaluacja, co oznacza, że argumenty funkcji są wyznaczane przed jej uruchomieniem, a do funkcji trafiają wyłącznie wyniki ewaluacji argumentów. W programie R jest inaczej.

Prześledźmy poniższy przykład użycia funkcji, która na pierwszy rzut oka jest identycznością. Na początku zdefiniujemy funkcję, której wynikiem jest pierwszy argument. Zainicjujemy zmienną globalną wartością 2.

```
prawieIdentycznosc <- function(x) {
  zmiennaGlobalna <<- 3
  x
}
zmiennaGlobalna <- 2
```

Czy tego wyniku oczekiwaliśmy?

```
prawieIdentycznosc(zmiennaGlobalna)
## [1] 3
```

W przykładzie wywoływana jest funkcja z argumentem `zmiennaGlobalna`. Wartość tego argumentu nie jest wyznaczana, aż do ostatniej linii kodu funkcji, ponieważ nigdzie wcześniej zmienna `x` (argument) nie jest użyta. Dlatego wynikiem funkcji wywołanej z argumentem równym 2 jest wartość 3, pomimo że argument funkcji `prawieIdentycznosc()` stał się trójką dopiero wewnątrz tej funkcji.

Wynik byłby inny, gdybyśmy wcześniej wymusili wyznaczenie wartości argumentu. W przykładzie poniżej w drugiej linii wyznaczana jest wartość `x`, przez co zmiana wartości `zmiennaGlobalna` w linii trzeciej nie wpływa na wynik.

```
prawieIdentycznosc <- function(x) {
   x = x + 0
   zmiennaGlobalna <<- 3
   x
}
zmiennaGlobalna  <- 2
prawieIdentycznosc(zmiennaGlobalna)
## [1] 2
```

Tym razem wynikiem funkcji `prawieIdentycznosc()` jest 2, zgodnie z oczekiwaniami, pomimo iż na pierwszy rzut oka instrukcja `x = x + 0` nie powinna wpłynąć na wynik.

Nie dość, że argumenty funkcji nie są wyznaczane bez potrzeby, to jeszcze funkcja ma pełną informację o tym, jakie wyrażenia zostały podane jako argumenty. Poniżej umieszczamy przykład, w którym wyrażenie będące argumentem funkcji odczytujemy i wypisujemy na ekranie bez wyznaczania jego wartości.

Funkcja `substitute()` przekształca wyrażenie w opis wyrażenia (obiekt klasy `call`). Funkcja `deparse()` przekształca dalej ten opis w napis, który można wyświetlić na ekranie lub użyć do innych celów.

W poniższej funkcji używając poleceń `deparse()` i `substitute()` wyświetlamy co zostało podane jako argument funkcji. Funkcja `printexpr()` wypisuje wyrażenie, które jest argumentem.

```
printexpr <- function(wyrazenie){
   cat(deparse(substitute(wyrazenie)))
}
printexpr(3+2 - log(14))
## 3 + 2 - log(14)
```

Na koniec jeszcze jeden przykład leniwej ewaluacji. W tym przykładzie użyjemy leniwego wartościowania do podania jako wartość domyślna wyrażenia, które wyznaczane będzie dopiero wewnątrz funkcji.

W poniższej funkcji argument `y` za wartość domyślną przyjmuje wyrażenie `z^2`, które jest określone dopiero wewnątrz ciała funkcji. W przestrzeni w której funkcja `wypiszKwadraty()` jest wywoływana zmienna `z` może nie istnieć. Gdyby wartościowanie było gorliwe, zwrócony byłby błąd dotyczący nieokreślonej wartości `z`. Ale czy tak będzie?

```
wypiszKwadraty <- function(x, y = z^2) {
  z <- 1:x
  print(y)
}
```

Zobaczmy co się stanie jeżeli podamy tylko pierwszy argument.

```
wypiszKwadraty(6)
## [1]  1  4  9 16 25 36
```

Błąd nie został zgłoszony. Dlaczego? Wartość `y` jest wyznaczana dopiero w drugiej linii, a do tego czasu już było wiadomo czym jest zmienna `z`.

Co się stanie jeżeli podamy oba argumenty? Funkcja działa tak jak oczekujemy.

```
wypiszKwadraty(6, 1:5)
## [1] 1 2 3 4 5
```

Zauważmy jednak, że nie możemy wywoływać funkcji `wypiszKwadraty` w poniższy sposób. Zmienna `z` w wywołaniu jest w innej przestrzeni nazw niż zmienna `z` wewnątrz funkcji `wypiszKwadraty`.

```
wypiszKwadraty(6, z^3)
## Error in print(y) : object "z" not found
```

### 1.6.4 Zarządzanie obiektami w przestrzeni nazw

Przestrzeń nazw to zbiór symboli do których przypisane są wartości. Można też o przestrzeni nazw myśleć jak o zbiorze/worku nazw zmiennych, które zostały zadeklarowane na tym samym poziomie (przestrzeni nazw). Przestrzenie nazw tworzą drzewiastą strukturę, każda z przestrzeni nazw ma zadeklarowaną nadrzędną przestrzeń nazw. Prawie każda, ponieważ na szczycie tego drzewa znajduje się przestrzeń nazw pusta `R_EmptyEnv`, która nie zawiera żadnego żadnego symbolu. Do nadrzędnej przestrzeni nazw można odwołać się funkcją `parent.env()`.

Jeżeli w pewnym miejscu program R szuka wartości dla określonego symbolu, szuka tej wartości najpierw w lokalnej przestrzeni nazw, jeżeli jej nie znajduje to szuka w przestrzeni nazw nadrzędnej i tak dalej aż znajdzie szukany symbol lub trafi do przestrzeni `R_EmptyEnv` (co kończy się komunikatem o błędzie, dana zmienna nie została znaleziona).

Główną przestrzenią nazw jest przestrzeń o nazwie `.GlobalEnv`, gdzie znajdują się nazwy zmiennych, które określiliśmy globalnie. Jeżeli włączamy/ładujemy nowy pakiet funkcją `library()`, przestrzeń nazw włączanego pakietu staje się przestrzenią nadrzędną dla przestrzeni globalnej. Używając w globalnej przestrzeni nazw symboli zdefiniowanych w załadowanych pakietach, nie powoduje błędu, bo dane symbole są widoczne, są bowiem w przestrzeni nazw nadrzędnej do globalnej.

Tworząc funkcję, tworzy się nową przestrzeń nazw a przestrzeń aktualną przypisuje się jako przestrzeń nadrzędną do przestrzeni nazw funkcji. Zmienne określone/zdefiniowane wewnątrz funkcji nie są widziane poza tą funkcją, mówimy wtedy, że są w innej przestrzeni nazw. Wewnątrz funkcji możemy odwoływać się do zmiennych znajdujących się w przestrzeni nazw tej funkcji (a więc zadeklarowanych wewnątrz tej funkcji) oraz do zmiennych z nadrzędnych przestrzeni nazw (w tym z głównej przestrzeni nazw).

W pakiecie R jest kilka funkcji, które pozwalają na zarządzanie przestrzeniami nazw. Najprostszym przykładem jest funkcja `ls()`, która wyświetla nazwy obiektów w aktualnej przestrzeni nazw (zachowanie domyślne) lub w przestrzeni wskazanej przez argument `name`, `pos` lub `envir` tej funkcji. Argumentem funkcji `ls()` jest też `all.names`, domyślnie argument `all.names` jest ustawiony na `FALSE`, co powoduje, że nazwy obiektów rozpoczynające się od znaku `.` nie są wyświetlane.

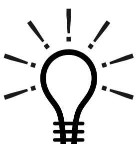
Podobne działanie mają funkcje `objects()` oraz `ls.str()`. Funkcja `ls.str()` wyświetla listę nazw zmiennych w aktualnej przestrzeni nazw oraz dodatkowo wyświetla informacje o strukturze poszczególnych zmiennych (strukturę jednego obiektu opisuje funkcja `str()`). Funkcja `lsf.str()` wyświetla listę funkcji widocznych w przestrzeni nazw.

Aby usunąć z pamięci operacyjnej wybrany (już niepotrzebny) obiekt możemy wykorzystać funkcję `rm()`. Powoduje ona usunięcie obiektu lub obiektów o nazwach podanych jako argument tej funkcji oraz zwolnienie pamięci zajmowanej przez te zmienne. Podobny efekt można uzyskać przypisując do już niepotrzebnej zmiennej wartość `NULL`. W tym przypadku pamięć zostanie zwolniona, ale nazwa zmiennej wciąż będzie widoczna w przestrzeni nazw.

Obiekty z pamięci operacyjnej nie są natychmiast usuwane. Dokładniej rzecz biorąc funkcja `rm()` markuje wskazane obiekty jako gotowe do usunięcia. Fizyczne zwolnienie pamięci zajmowanej przez ten obiekt następuje dopiero po uruchomienia „śmieciarza", czyli mechanizmu Garbage Collection. Zazwyczaj ten mechanizm jest przez R uruchamiany automatycznie, gdy tylko zachodzi taka potrzeba. Można również ten proces uruchomić ręcznie wywołując funkcję `gc()`.
Aby usunąć wszystkie zmienne z przestrzeni roboczej można użyć polecenia `rm(list = ls())`.

Funkcją `save.image()` można zapisać do pliku wszystkie zmienne znajdujące się w aktualnej (lub wskazanej argumentem `envir`) przestrzeni nazw. Jest to przydatne polecenie, gdy chcemy zachować aktualny stan pracy i np. przesłać go współpracownikom. Wybraną zmienną lub zmienne możemy zapisać poleceniem `save()`. Funkcja `savehistory()` zapisuje historię wszystkich wykonanych poleceń do wskazanego pliku. Dodatkowe informacje o stanie środowiska R, np. lista załadowanych aktualnie pakietów, można uzyskać uruchamiając funkcję `sessionInfo()`.

Nazwy zmiennych mogą składać się z dużych i małych liter, cyfr i znaków `'_'` oraz `'.'`. Operatory arytmetyczne i spacje w nazwach zmiennych są niedozwolone (ponieważ byłyby niejednoznacznie traktowane przez parser języka). Jeżeli chcemy jakiś napis zawierający niedozwolone znaki zamienić na poprawną nazwę zmiennych możemy skorzystać z funkcji `make.names()`. Usuwa ona niedozwolone znaki z łańcucha znaków będącego jej argumentem.



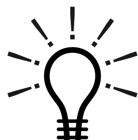
Oczywiście, powinniśmy pamiętać, że w programie R nie ma rzeczy niemożliwych i wyjątkowo uparte osoby mogą korzystać z najdziwniejszych możliwych nazw zmiennych. Poniżej znajduje się przykład, jak zdefiniować i używać zmiennej o nazwie `a\a`. Można to zrobić korzystając z funkcji `assign()` i `get()`.
Ze zrozumiałych powodów nie polecam tworzenia takich nazw. Ale poniższy przykład może się przydać w sytuacjach, gdy R wygeneruje automatycznie jakąś bardzo dziwną nazwę dla zmiennej, np. nazwą będzie ścieżka do jakiegoś pliku. Jak się taką zmienną posługiwać? Zobaczmy.

Zaczynamy od przypisania wartości do zmiennej o dziwnej nazwie a następnie sprawdźmy czy ta zmienna jest w pamięci programu R.

```
assign("a\a", 3)
ls()
## [1] "a\a"
```

Odczytajmy jej wartość.

```
get("a\a")
## [1] 3
```



Napisaliśmy, że tworząc funkcje tworzymy nową przestrzeń nazw. Taka przestrzeń nazw nie ma nazwy, ale jest widoczna przez funkcje, które w niej powstały. Dzięki temu mechanizmowi można zamykać zbiory zmiennych w przestrzenie nazw, niewidoczne „z zewnątrz".

Prześledźmy poniższy, bardzo ciekawy przykład. Zmienna `rejestr` otrzymuje wartość zwróconą przez wywołanie anonimowej funkcji. Ta anonimowa funkcja jako wynik zwraca trzy funkcje o nazwach `get`, `set` i `inc`, które operują na zmiennej v widocznej w przestrzeni nazw anonimowej funkcji. Czyli te trzy funkcje współdzielą przestrzeń nazw, nadrzędną do ich przestrzeni nazw, zawierającą obiekt `licznik`.

```
rejestr <- {function () {
           licznik <- 0
           get <- function () licznik
           set <- function (x) licznik <<- x
           inc <- function () licznik <<- licznik + 1
           list(get=get, set=set, inc=inc)
           }}()
```

Wartość zmiennej `licznik` jest trudno dostępna, nie będzie widziana w globalnej przestrzeni nazw, a ślad po przestrzeni nazw funkcji anonimowej jest już tylko w nadrzędnych przestrzeniach nazw funkcji `get`, `set` i `inc`.

```
rejestr$set(10)
rejestr$get()
## [1] 10
```

Możemy teraz kopiować wartość zmiennej `rejestr`, wszystkie kopie będą miały dostęp do tej samej przestrzeni nazw z tą samą zawartością. Dzięki temu, mamy dosyć szczelnie kontrolowany sposób dostępu do wartości zmiennej `licznik`.

```
rejestr2 <- rejestr
rejestr2$get()
## [1] 10
```

Zarówno funkcje w zbiorze rejestr, jak i rejestr2 odwołują się do tej samej przestrzeni nazw.

```
rejestr2$inc()
rejestr$get()
## [1] 11
```

### 1.6.5 Wprowadzenie do grafiki

Soon, they'll be speaking R on the subway.

Michael Rennie
fortune(68)

Pisaliśmy już wiele o tym, że możliwości programu R w zakresie graficznego prezentowania danych są olbrzymie. Czas na poznanie ich bliżej. Znacznie dokładniej o grafice napiszemy w rozdziale 4, tutaj zasygnalizujemy możliwości. Wykresy w programie R tworzyć można korzystając z wielu różnych funkcji, ale na początku najpopularniejszym sposobem jest skorzystanie z funkcji `plot()` (jest to też jedna z najczęściej przeciążanych funkcji, co oznacza, że ma bardzo wiele wyspecjalizowanych implementacji).

Wprowadzenie do grafiki zacznijmy od prostego przykładu. Przygotowujemy siatkę punktów a następnie narysujemy na niej wartości funkcji $\sin(x)$. Później dorysowujemy do niej wartości funkcji $cos(x)$.

```
x <- seq(-2*pi,2*pi,by=0.3)
plot(x, sin(x), type="b",main="Wykres funkcji sin(x) i cos(x)",col="red")
lines(x, cos(x), col="blue", type="l")
```

W pierwszej linii powyższego przykładu tworzony jest wektor liczb z przedziału $-2\pi..2\pi$ z użyciem funkcji `seq()`. Druga linijka, to wywołanie funkcji `plot()`. Funkcja ta przygotowuje okno graficzne do narysowania wykresu, czyści jego poprzednią zawartość, inicjuje osie, ustawia układ współrzędnych, określa rozmiary marginesów i robi wiele innych rzeczy (więcej szczegółów przedstawionych będzie w podrozdziale poświęconym zaawansowanej grafice). Po zainicjowaniu okna graficznego funkcja `plot()` narysuje linię łamaną łączącą punkty o współrzędnych `x,y` wskazanych przez jej dwa pierwsze argumenty (czyli wektor `x` i wartość funkcji sinus w tych punktach). Gdyby podany był tylko jeden argument, to będzie on uznany za wektor współrzędnych `y` a za wektor `x` użyty będzie wektor kolejnych liczb naturalnych.

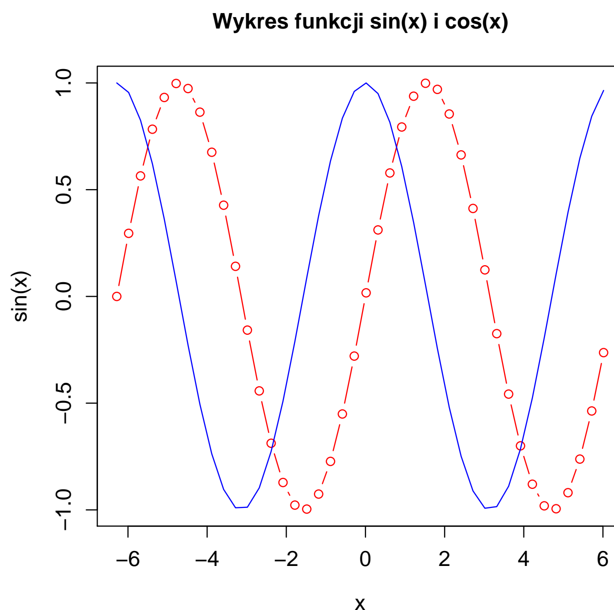
W powyższym przykładzie funkcja `plot()` rysuje kolejne punkty, a następnie łączy je linią. Odpowiedzialny za to postępowanie jest argument `type="b"` tej funkcji. Wartość `"b"` oznacza, że chcemy rysować zarówno linie, jak i punkty (skrót od **b**oth). W trzeciej linii przykładu wartość `type="l"` oznacza, że rysowane mają być wyłącznie linie. Inne możliwe wartości tego argumentu, to **p** (punkty), **n** (nic), **s** (schodki), **h** (pionowe kreski, podobne do histogramu). Argument `col` funkcji `plot()` umożliwia wskazanie koloru w jakim ma być narysowana nowa linia. Argument `main` pozwala na określenie, jaki napis ma być narysowany jako tytuł wykresu. Więcej o argumentach funkcji graficznych znaleźć można w podrozdziale 4.2. Efekt działania powyższych poleceń znajduje się na rysunku 1.3.

Funkcje matematyczne można rysować korzystając z polecenia `curve()`. Pierwszym argumentem jest funkcja lub wyrażenie w którym występuje zmienna `x`. Wynikiem funkcji `curve()` będzie wykres funkcji lub wyrażenia wskazanego pierwszym argumentem w przedziałach określonych przez kolejne dwa argumenty.

Poniżej dwa przykłady do samodzielnego przećwiczenia.

```
curve(sin, from = -2*pi, to = 2*pi)
curve(x^2 - sin(x^2), -2, 2)
```

Kolejną przydatną funkcją graficzną jest `abline()`. Pozwala ona na dorysowanie linii prostej podając jako argumenty współczynniki równania prostej, czyli




**Rysunek 1.3:** Wykres narysowany różnymi rodzajami linii

równania `y = ax + b`. Jeżeli chcemy narysować linię poziomą lub pionową, to wystarczy podać tylko jedną współrzędną, odpowiednio określając argument `h` dla linii poziomych lub `v` dla linii pionowych.

Poniższe polecenie nic nie narysuje, przygotuje jedynie okno graficzne i układ współrzędnych.

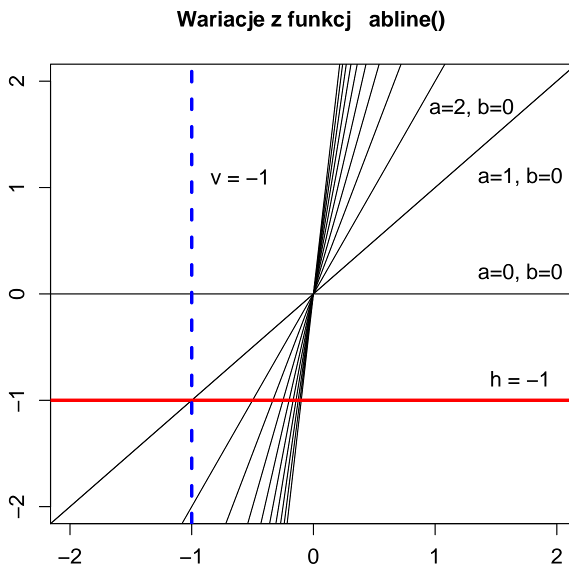
```
plot(0, xlim=c(-2,2), ylim=c(-2,2), type="n", xlab="", ylab="",
                main="Wariacje nt. funkcji abline()")
```

W kolejnych liniach przedstawiono kilka przykładowych wywołań funkcji `abline()`. Wynik działania tego kodu jest umieszczony na rysunku 1.4. Za argument funkcji `abline()` można również podać model liniowy otrzymany z użyciem `lm()`. W tym przypadku do wykresu dorysowanych będzie kilka prostych regresji określonych przez równanie `y = a x + b` a następnie kilka linii poziomych i pionowych.

```
abline(0, 0)
for (i in 1:10)
     abline(0, i)
abline(h=-1, lwd=3, col="red")
abline(v=-1, lwd=3, lty=2, col="blue")
```

Na wykres nanosimy jeszcze kilka napisów.

```
text( 1.7, 0.2, "a=0, b=0")
text( 1.7, 1.1, "a=1, b=0")
text( 1.3, 1.7, "a=2, b=0")
text( 1.7,-0.8, "h = -1")
text(-0.6, 1.1, "v = -1")
```




**Rysunek 1.4:** Przykładowe użycia funkcji `abline()`

Funkcja `text()` służy do umieszczania napisów na wykresach. Kolejne argumenty tej funkcji to: współrzędne punktu, w którym ma znaleźć się napis oraz łańcuch znaków określający, co ma być do wykresu dopisane. Dostępnych argumentów funkcji graficznych jest znacznie więcej, o czym łatwo się przekonać przeglądając pomoc do wymienionych funkcji. Wrócimy do tego tematu w podrozdziale 4.2.

Funkcja `abline()` dorysowuje linie do istniejącego wykresu (inaczej niż funkcja `plot()`, która tworzy nowy wykres). Przed jej wywołaniem należy więc zapewnić, by jakieś okno graficzne z wykresem było aktywne i zainicjowane. W powyższym przykładzie wykorzystaliśmy do tego funkcje plot, z argumentem `type="n"`. Ten argument wyłącza rysowanie wykresu. W tym przypadku funkcja `plot()` jedynie zainicjowała okno graficzne, osie oraz opis osi i wykresu nic jednak nie rysując.
Do funkcji `abline()`, podobnie jak do większości funkcji graficznych, można podawać takie same argumenty, jak w przypadku funkcji `plot()`, czyli np. argumenty `col`, `lwd` lub `lty`. Argument `lty` odpowiada za styl linii, `lwd` za jej grubość. Więcej informacji o parametrach graficznych znaleźć można w podrozdziale 4.2.8.

### 1.6.6 Wprowadzenie do operacji na plikach i katalogach

Zanim zaczniemy omawiać funkcje służące do odczytywania i zapisywania danych z plików, przedstawimy kilka funkcji do wykonywania podstawowych operacji na katalogach i plikach.

Zacznijmy od dwóch funkcji: `getwd()` i `setwd()`. Pierwsza z tych funkcji sprawdza, jaki katalog na dysku jest aktualnie katalogiem roboczym, druga pozwala na zmianę katalogu roboczego. Podobny efekt można uzyskać wyklikując opcję `Change dir` w menu. Zmiana katalogu roboczego na początku pracy pozwala na znaczne skrócenie zapisu ścieżek do plików, ponieważ wszystkie ścieżki do plików możemy podawać w postaci bezwzględnej (niewygodne) i względnej, względem katalogu roboczego (wygodne).

```
setwd("c:/Projekty/Dane")
getwd()
## [1] "c:/Projekty/Dane"
```

Zobaczmy co jest w tym katalogu.

```
lista.plikow <- list.files()
lista.plikow
## [1] "daneFWF.txt"        "daneMatlab.MAT"     "daneMieszkania.csv"
## [4] "daneSAS.sas7bdat"   "daneSoc.csv"        "daneSPSS.sav"
```

Aby wyświetlić listę plików znajdujących się w aktualnym (lub innym wskazanym) katalogu można posłużyć się funkcją `list.files()` lub `dir()`. Pierwszym, opcjonalnym, argumentem tych funkcji jest ścieżka do katalogu, którego zawartość chcemy wyświetlić (domyślnie jest to aktualny katalog roboczy). W tabeli 1.7 przedstawiono listę funkcji do operowania na plikach, a poniżej przedstawiono przykład wywołania wybranych funkcji z tej listy.

Pomocne funkcje do tymczasowego składowania danych w plikach to `tempdir()` i `tempfile()`. Wynikiem pierwszej z nich jest ścieżka do tymczasowego, nieistniejącego jeszcze katalogu. Wynikiem drugiej funkcji jest ścieżka do nieistniejącego jeszcze pliku (nazwa jest losową kombinacje znaków), który może być wykorzystany jako plik tymczasowy. Obie funkcje są przydatne, jeżeli chcemy przechować tymczasowo pewne dane w plikach, ale nie mamy pomysłu na ich nazwę.

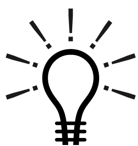
Na przykładzie poniżej generujemy losową nazwę dla tymczasowego pliku. Wpisujemy do tego pliku napis `"Poczatek pliku"` a na koniec kasujemy tymczasowy plik.

```
tmpf <- tempfile()
cat(file=tmpf, "Poczatek pliku")
#
# Tu mozna wykonywac operacje na pliku
# ...
unlink(tmpf)
```

Kolejną przydatną funkcją do operacji na plikach jest `sink()`. Zapisuje ona do wskazanego pliku tekstowego przebieg interakcji z programem R. Jej wywołanie powoduje przekazanie wyjścia z konsoli programu R do wskazanego pliku (domyślnie wyjście jest kierowane zarówno do pliku jak i na ekran, ale można zarządzać, by było kierowane tylko do pliku). W wyniku jej działania, we wskazanym pliku znajduje się cała historia wykonanych poleceń wraz otrzymanymi wynikami, czyli kopia wszystkiego co działo się na konsoli programu R.

**Tabela 1.7:** Funkcje do operacji na plikach z pakietu base

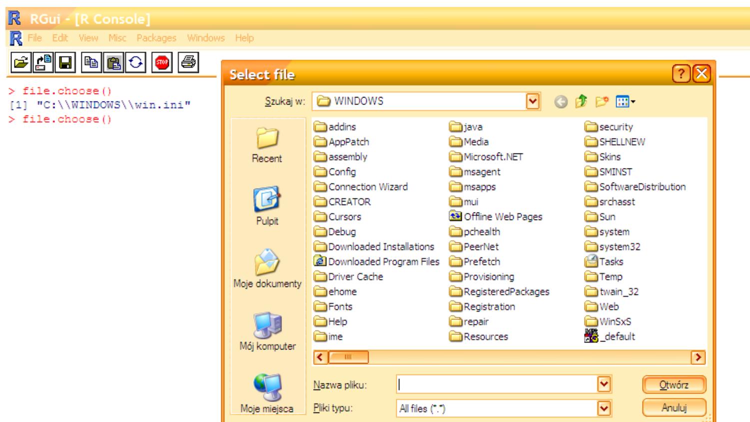
| | |
|---|---|
| `file.create(...)` | Funkcja tworzy pliki o zadanych nazwach, jeżeli takie pliki istnieją, to ich zawartość jest kasowana. |
| `file.exists(...)` | Funkcja sprawdza czy pliki o zadanych nazwach istnieją. |
| `file.remove(...)` | Funkcja usuwa pliki o zadanych nazwach (patrz też funkcja `unlink()`). |
| `file.rename(from, to)` | Funkcja zmienia nazwę pojedynczego pliku. |
| `file.append(file1, file2)` | Funkcja dokleja plik o nazwie `file2` do pliku `file1`. |
| `file.copy(from, to, overwrite=FALSE)` | Funkcja do kopiowania pliku `from` w pozycje wskazaną przez argument `to`. |
| `file.symlink(from, to)` | Funkcja do tworzenia linków symbolicznych (pod Unixami). |
| `dir.create(path, showWarnings=TRUE, recursive=FALSE)` | Funkcja do tworzenia katalogów. Wynikiem tej funkcji jest wartość `TRUE`, jeżeli operacja utworzenia katalogu została wykonana pomyślnie. Jeżeli dany katalog już istnieje lub nie udało go się utworzyć funkcja przekazuje wartość `FALSE`. |
| `unlink(x, recursive=F)` | Funkcja do usuwania plików lub katalogów. |
| `file.info(...)` | Wynikiem tej funkcji są informacje o wskazanych plikach. |
| `file_test(op, x, y)` | Funkcja do testowania plików. Dostępne testy to: jednoargumentowe (tylko `x` jest używane) `op="-f"` istnieje i nie jest katalogiem, `op="-d"` istnieje i jest katalogiem oraz dwuargumentowe `op="-nt"` jest młodszy niż (pod uwagę brane są daty modyfikacji) i `op="-ot"` jest starszy niż. |
| `file.show(...)` | Funkcja wyświetla w oknie R zawartość jednego lub większej liczby plików. |



Jeżeli nie chcemy z klawiatury wpisywać ścieżki do pliku, to możemy wyklikać ją korzystając z funkcji `file.choose()` lub `choose.files()`. Obie funkcje otwierają okno systemowe pozwalające na wskazanie pliku lub plików. Wynikiem obu funkcji jest wektor ścieżek do wskazanych przez użytkownika plików.
Przypomnijmy też, że program R ma możliwość uzupełniania ścieżek. Jeżeli przy wpisywaniu ścieżki do pliku lub katalogu naciśniemy `Tab`, to program R uzupełni nazwę wpisywanego katalogu lub pliku. Jeżeli ścieżkę można uzupełnić na wiele sposobów, to R wyświetla listę wszystkich możliwości.

### Odczytywanie i zapisywanie plików tekstowych

W programie R jest wiele różnych funkcji umożliwiających czytanie danych z pliku i zapisywanie danych do pliku. Funkcje te mają wiele argumentów pozwalających na określenie rodzaju kodowania, znaku separatora, kropki dziesiętnej, typu od-




**Rysunek 1.5:** Okno systemowe umożliwiające wskazanie jednego lub większej liczby plików. Wynik działania funkcji `file.choose()` lub `choose.files()`

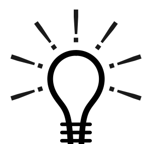
czytywanych danych i innych detali opisujących format danych w pliku. Aby nie utonąć w szczegółach, poniżej przedstawione są jedynie najczęstsze przykłady użycia. Wymienione funkcje mają bardzo dokładnie opracowane pliki pomocy, tam zainteresowani oraz potrzebujący znajdą z pewnością więcej informacji.

Analizując dane najczęściej korzysta się z danych zapisanych w strukturze tabelarycznej i w takiej postaci przechowuje się te dane w plikach tekstowych. Do operacji na plikach tekstowych, w których są dane zapisane w tej postaci wykorzystać można funkcje `read.table()` i `write.table()`. Obie funkcje są szczegółowo opisane w podrozdziale 2.4.1, w tym rozdziale przedstawiamy jedynie krótkie wprowadzenie. Zacznijmy od prostego przykładu wczytania danych z pliku.

```
macierz <- read.table("nazwa.pliku.z.danymi")
```

W pliku `"nazwa.pliku.z.danymi"` mogą być nazwy kolumn (zmiennych) i/lub wierszy (przypadków). Kolejne wartości w pliku rozdzielane mogą być różnymi znakami (najpopularniejsze separatory to przecinek, średnik, spacja lub tabulacja). Poniższe polecenie odczytuje dane z pliku, w którym w pierwszej linii umieszczone są nazwy kolumn a wartości kolejnych pól rozdzielane są znakami tabulacji.

```
macierz <- read.table("nazwa.pliku", header=TRUE, sep="\t")
```



Ścieżka do pliku może być również adresem URL! Korzystając z takich ścieżek możemy odczytywać dane z plików umieszczonych na innych komputerach. Przykładowe zbiory danych wykorzystywane w tej książce mogą być w ten sposób bezpośrednio odczytane z Internetu.

Wartość lub wektor wartości możemy zapisać do pliku korzystając z funkcji `cat()`. Domyślnie wynik tej funkcji jest wyświetlany w konsoli, ale zmieniając wartość argumentu `file` możemy zapisać wyniki do pliku. Tak jak na poniższym przykładzie.

```
cat(wektor, file="nazwa.pliku", append=FALSE)
```

Argument append określa, czy wynik tej funkcji ma być dopisany do końca pliku (o ile plik istnieje), czy też ten wynik ma nadpisać ewentualną zawartość wskazanego pliku. Jeżeli wskazany plik nie istnieje, wynik jest w obu przypadkach taki sam. Do zapisu danych (wektora, macierzy lub ramki danych) w formacie tabelarycznym można wykorzystać funkcję `write.table()`. Poniższe polecenie zapisuje dane rozdzielając kolejne elementy znakiem tabulacji.

```
write.table(macierz, file="nazwa.pliku", sep="\t")
```

Duże i złożone obiekty lepiej przechowywać w postaci binarnej. Zapis w formacie plików binarnych umożliwia funkcja `save()`. Funkcja ta zapisuje wskazany obiekt lub listę obiektów w formacie `Rdata` (zazwyczaj pliki z rozszerzeniem `Rdata` lub `rda`). Do takiego formatu można zapisać nie tylko liczby, ale też złożone obiekty i funkcje. Jeżeli chcemy zapisać do pliku wartość wszystkich obiektów z przestrzeni nazw, to można skorzystać z funkcji `save.image()`. Zapisuje ona do pliku wszystkie dostępne obiekty. Podobny efekt ma polecenie z menu `File / Save workspace`.

## 1.7 Zadania do części „Łagodne wprowadzenie...”

W tym rozdziale przedstawiamy zbiór zadań do samodzielnego wykonania. Zadania zostały podzielone na trzy poziomy trudności, oznaczane liczbą liter `R`

R zadania podstawowe, bazujące wyłącznie na materiale z tej książki (dla większości użytkowników jest to zakres wystarczający do codziennej pracy z programem R),

RR zadania o podwyższonej trudności, dla biegłych użytkowników, wymagają często poszukania dodatkowych informacji w sieci lub w plikach pomocy R,

RRR zadania typu „sprawdź się”, najczęściej można je rozwiązać na wiele sposobów, rzecz w tym, by wybrać ten najlepszy.

Pod adresem `http://www.biecek.pl/R/odpowiedzi.R` znajdują się przykładowe rozwiązania poniższych zadań. Zapraszamy czytelników do zgłaszania własnych ciekawych rozwiązań i/lub zadań.

R Zadanie 1.1

Skonstruuj wektor kwadratów liczb od 1 do 100. Następnie używając operatora dzielenia modulo i funkcji `factor()` zlicz, które cyfry oraz jak często występują na pozycji jedności w wyznaczonych kwadratach.

RR Zadanie 1.2

Zbuduj własne tablice trygonometryczne. Przygotuj ramkę danych, w których zebrane będą informacje o wartościach funkcji sinus, cosinus, tangens i cotangens dla kątów: $0^\circ$, $30^\circ$, $45^\circ$, $60^\circ$, $90^\circ$. Zauważ, że funkcje trygonometryczne w R przyjmują argumenty w radianach.

R Zadanie 1.3

Przygotuj wektor 30 łańcuchów znaków następującej postaci: `liczba.litera`, gdzie liczba to kolejne liczby od 1 do 30 a litera to trzy duże litery `A`, `B`, `C` występujące cyklicznie.

R Zadanie 1.4

Patrz załącznik z opisem zbiorów danych.

Wczytaj zbiór danych `dane0` i napisz funkcję lub pętlę sprawdzającą typ i klasę każdej kolumny tego zbioru.

R Zadanie 1.5

Z odczytanej ramki danych `dane0` wyświetl tylko dane z wierszy o parzystych indeksach.

R Zadanie 1.6

Używając operatorów logicznych wyświetl ze zbioru danych tylko wiersze odpowiadające: pacjentkom starszym niż 50 lat u których wystąpiły przerzuty do węzłów chłonnych (cecha `Wezly.chlonne=1`).

R Zadanie 1.7

Wyświetl nazwy kolumn w zbiorze danych `dane0`, a następnie oblicz długość (liczbę znaków) nazw kolejnych kolumn.

R Zadanie 1.8

Napisz funkcję, która za argumenty przyjmie wektor liczb, a jako wynik zwróci trzy najmniejsze i trzy największe liczby. Jeżeli wejściowy wektor jest krótszy niż trzy liczby, to wyświetlany powinien być napis „za krótki argument".

RR Zadanie 1.9

Zmodyfikuj funkcję z poprzedniego zadania tak, by otrzymywała też drugi argument `ile`, którym można określić liczbę skrajnych wartości wyznaczanych jako wynik. Domyślną wartością tego argumentu powinna być liczba `3`.

RR Zadanie 1.10

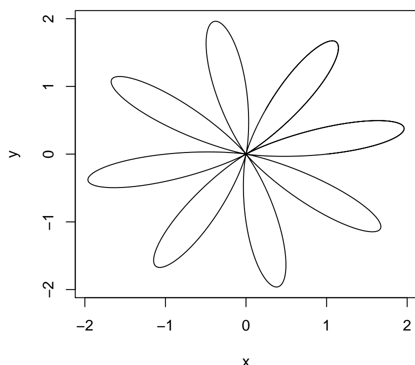
Napisz funkcję `poczatek()` przyjmującą za pierwszy argument wektor, macierz lub ramkę a za drugi argument liczbę `n`. Niech to będzie przeciążona funkcja. Dla wektora powinna ona w wyniku zwracać `n` pierwszych elementów, dla macierzy i ramki danych powinna zwracać podmacierz o wymiarach `n×n`.

Wskazówka: trzeba zamienić współrzędne na układ kartezjański przekształceniami $x = r\cos(\phi)$ i $y = r\sin(\phi)$. Autor upierał się, że tego uczą w gimnazjum i takie podpowiedzi są zbędne, ale prawda jest taka, że autor nigdy nie był w gimnazjum. Przyp. żony.

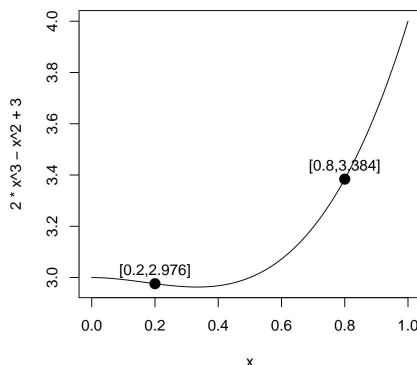
RR Zadanie 1.11 Narysuj funkcję, która w układzie biegunowym ma współrzędne:

$$\begin{aligned} r &= 1 + \sin(t), \\ \phi &= c * t. \end{aligned}$$

dla $c = 1$, $c = 0.1$ i $c = 2.2$. Zobacz przykład na rysunku 1.6.




**Rysunek 1.6:** Ilustracja do zadania 1.11




**Rysunek 1.7:** Ilustracja do zadania 1.12

R Zadanie 1.12

Używając instrukcji `curve()` narysuj wykres funkcji $f(x) = 2x^3 - x^2 + 3$ na przedziale $[0, 1]$.

Na wykresie tej funkcji zaznacz punkty odpowiadające wartościom w punktach 0.2 i 0.8, a następnie na wykres nanieś napisy opisujące współrzędne tych punktów. Zobacz przykład na rysunku 1.7.

R Zadanie 1.13

Pod adresem `http://www.biecek.pl/R/dane/daneBioTech.csv` znajduje się plik tekstowy z danymi. Dane są w formacie tabelarycznym, mają nagłówek, kolejne pola rozdzielane są średnikiem a kropką dziesiętną jest przecinek. Wczytaj te dane do programu R i przypisz je do zmiennej `daneBT`.

R Zadanie 1.14

Z odczytanych w poprzednim zadaniu danych wybierz tylko pierwsze trzy kolumny i pierwsze 10 wierszy. Zapisz ten fragment danych do pliku `maleDane.txt` na dysk `c:\` (użytkownicy Linuxa mogą zapisać do innego katalogu). Rozdzielaj kolejne pola znakiem tabulacji a kropką dziesiętną będzie kropka. Sprawdź w dowolnym edytorze tekstowym, co zapisało się do tego pliku.

RR Zadanie 1.15

Skonstruuj wektor 100 liczb, który jest symetryczny (tzn. elementy czytane od końca tworzą ten sam wektor co elementy czytane od początku). Pierwsze 20 liczb to kolejne liczby naturalne, następnie występuje 10 zer, następnie 20 kolejnych liczb parzystych (pozostałe elementy określone są przez warunek symetrii). Napisz funkcję, która sprawdza czy wektor jest symetryczny i sprawdź czy wygenerowany wektor jest symetryczny.

RRR Zadanie 1.16

Napisz funkcję `localMin()`, której argumentem będzie ramka danych, a wynikiem będą te wiersze, w których w jakiejkolwiek liczbowej kolumnie występuje wartość najmniejsza dla tej kolumny. Kolumny z wartościami nieliczbowymi nie powinny być brane pod uwagę.

Innymi słowy jeżeli ramka ma trzy kolumny z wartościami liczbowymi, to wynikiem powinny być wiersze, w których w pierwszej kolumnie występują wartości minimalne dla pierwszej kolumny oraz wiersze, w których w drugiej kolumnie występują wartości minimalne dla drugiej kolumny oraz wiersze, w których w trzeciej kolumnie występują wartości minimalne dla trzeciej kolumny. Odczytaj ramkę danych z zadania 1.13 i sprawdź działanie napisanej funkcji.

R Zadanie 1.17

Poniższa funkcja nie działa poprawnie, powinna wyznaczać kwadraty kolejnych liczb ale tego nie robi. Skopiuj ją do programu R a następnie użyj instrukcji `fix()`, by poprawić funkcję `kwadratyLiczb()`.

```
kwadratyLiczb <- function(x) {
 1:x^2
}
```

R Zadanie 1.18

Funkcja `ecdf()`, wyznacza dystrybuantę empiryczną. Przyjrzyj się trzeciej linii z poniższego przykładu oraz spróbuj przewidzieć co jest wynikiem tego wyrażenia i jaka funkcja jest wywoływana jako druga.

```
data(iris)
x <- iris[,1]
ecdf(x)(x)
```

R Zadanie 1.19

Znajdź liczbę `x` z przedziału [0 − 1] dla którego poniższe wyrażenie zwraca wartość `FALSE`.

```
x + 0.1 + 0.1 == 0.1 + 0.1 + x
```

R Zadanie 1.20

Dla zbioru danych `iris` narysuj wykres przedstawiający zależność pomiędzy dwoma wybranymi zmiennymi. Użyj funkcji `png()` i `pdf()` aby zapisać ten wykres do pliku.

# Rozdział 2

# pazuRrry

## 2.1 Typy zmiennych i operacje na nich

Ten rozdział poświęcony jest opisowi typów wykorzystywanych w programie R. Uzupełnimy tym samym podrozdział 1.5.5 opisując szczegółowiej tak typy, jak i typowe funkcje do typowej pracy z tymi typami.

> Does anyone know someone working in this area in France who is not a JARP (Just Another R Person)?
>
> Jan de Leeuw
> fortune(141)

### 2.1.1 Typ czynnikowy/wyliczeniowy

Typ czynnikowy nazywany jest też typem wyliczeniowym lub kategorycznym. Wartości typu czynnikowego tworzy się najczęściej z użyciem funkcji `factor()`, której deklaracja jest następująca:

```
factor(x=character(),levels=sort(unique.default(x),na.last=TRUE),
       labels=levels, exclude=NA, ordered=is.ordered(x))
```

Wartość argumentu `x` tej funkcji zostanie zamieniona na wartość typu czynnikowego. Jeżeli pierwszym argumentem jest zmienna typu czynnikowego, to dodatkowym efektem będzie pominięcie poziomów, które w danym wektorze nie występują, ale są wymienione we właściwości `levels`. Często po wczytywanie danych z innego pakietu otrzymuje się zmienne z określonymi poziomami, nawet jeżeli nie występują one w bazie danych.

Zmienne czynnikowe dzieli się na dwie grupy: na takie, w których poziomy nie są ze sobą w żadnej relacji porządku i na takie, w której poziomy są w pewnej logicznej kolejności. Dla wielu zmiennych takiego porządku nie da się wprowadzić, bo jak np. określić porządek dla państw, dla owoców lub leków. Dla pewnych poziomów taki porządek wynika z interpretacji poziomów, np. poziomy `"duży"`, `"średni"`, `"mały"` są w naturalnym, logicznym porządku. Aby zaznaczyć, że poziomy są uporządkowane należy w funkcji `factor()` argument `ordered` ustawić na `TRUE`. W rozdziale poświęconym statystyce pokażemy, że w zależności od tego, czy poziomy są uporządkowane, czy nie, możemy wykonywać różne analizy, dla typu uporządkowanego stosowane są inne kontrasty do wyznaczania zmiennych pustych, niż dla zmiennych czynnikowych.

Domyślnie, etykiety zostaną utworzone na podstawie konwersji elementów wejściowego wektora na typ znakowy, ale można też podać wektor etykiet, które mają

być użyte jako nazwy poziomów (argument `labels`). Można też wskazać wartości, które mają zostać wyłączone z etapu tworzenia etykiet (argument `exclude`). Te wartości zostaną zamienione na `NA`.

Na potrzeby kolejnych przykładów przygotujmy wektor napisów i zamieniamy go na wektor zmiennych czynnikowych.

```
wektor <- c("Wroclaw", "Poznan", "Wroclaw", "Czestochowa", "Wroclaw",
            "Wroclaw")
(wyliczeniowa <- factor(wektor))
## [1] Wroclaw    Poznan     Wroclaw    Czestochowa Wroclaw    Wroclaw
## Levels: Czestochowa Poznan Wroclaw

factor(wyliczeniowa, exclude="Poznan")
## [1] Wroclaw    <NA>       Wroclaw    Czestochowa Wroclaw    Wroclaw
## Levels: Czestochowa Wroclaw
```

Zmieniamy etykiety dla elementów wektora zmiennej wyliczeniowej.

```
factor(wyliczeniowa, labels = c("miasto 1","miasto 2","miasto 3"))
## [1] miasto 3 miasto 2 miasto 3 miasto 1 miasto 3 miasto 3
## Levels: miasto 1 miasto 2 miasto 3
```

Funkcja `levels()` jako wynik zwraca nazwy poziomów zmiennej typu wyliczeniowego. Można jej również użyć do modyfikacji tych nazw, korzystając z konstrukcji `levels() <-` (patrz przykład poniżej). Funkcja `nlevels()` jako wynik przekazuje liczbę poziomów wskazanej zmiennej. W naszym przykładzie wynikiem tej funkcji jest trzyelementowy wektor etykiet.

```
levels(wyliczeniowa)
## "Czestochowa" "Poznan" "Wroclaw"
```

Zmieniamy etykiety zmiennych.

```
levels(wyliczeniowa) <- c("Medaliki", "Pyrlandia", "Miasto spotkan")
summary(wyliczeniowa)
##       Medaliki      Pyrlandia Miasto spotkan
##              1              1              4
```

Liczba poziomów dla tej zmiennej.

```
nlevels(wyliczeniowa)
## [1] 3
```

Dla zmiennych wyliczeniowych funkcją `table()` można wyznaczać macierz kontyngencji, czyli macierz zliczającą liczności dla poszczególnych kombinacji czynników. Macierze kontyngencji można wyznaczać dla dowolnej liczby zmiennych. Wynikiem funkcji `table()` są jednowymiarowe, dwuwymiarowe lub wielowymiarowe macierze (w zależności od liczby zmiennych).

Jeżeli dane nie są przechowywane w wektorach, ale w ramce danych, to do wyznaczenia macierzy kontyngencji można użyć funkcji `xtabs()`. Pierwszym argumentem tej funkcji jest formuła opisująca, które kolumny wejściowej ramki danych mają być użyte do wyznaczenia macierzy kontyngencji. W sytuacji, gdy rozważamy wiele zmiennych czynnikowych wynik działania funkcji `table()` i `xtabs()` jest mało czytelny (trudno przedstawić np. pięciowymiarową macierz). Aby poprawić

czytelność można użyć funkcji `ftable()`, której wynikiem jest płaska, dwuwymiarowa macierz kontyngencji. Do analiz macierzy kontyngencji przydatna jest też funkcja `margin.table()` wyznaczająca brzegowe liczebności oraz funkcja `prop.table()` wyznaczająca częstości (frakcje) zamiast liczebności. Używając funkcji `addmargins()` do macierzy kontyngencji można dodać kolumnę i wiersz z brzegowymi liczebnościami.

Do zamiany zmiennej liczbowej na zmienną przedziałową (czynnikową) możemy wykorzystać funkcję `cut()` o argumentach `x` i `breaks`. Elementy wektora `x` są zamieniane na wartości poziomów w zależności od tego, do jakiego przedziału określonego przez wektor `breaks` należą. Dodatkowy argument `labels` pozwala na określenie etykiet dla tak otrzymanych czynników.

```
wektor <- 1:10
(czynnik1 <- cut(wektor,c(0,5,11)))
##  [1] (0,5]  (0,5]  (0,5]  (0,5]  (0,5]  (5,11] (5,11] (5,11] (5,11]
    (5,11]
## Levels: (0,5] (5,11]
```

Do kolejnych przykładów potrzebujemy jeszcze zmiennej czynnikowej.

```
(czynnik2 <- cut(wektor^2, c(0,50,100)))
##  [1] (0,50] (0,50] (0,50] (0,50] (0,50] (0,50] (0,50] (50,100]
    (50,100] (50,100]
## Levels: (0,50] (50,100]
```

Funkcją `table()` możemy wyznaczyć macierz kontyngencji/tabelę krzyżową.

```
(tabela <- table(czynnik1, czynnik2))
##         czynnik2
## czynnik1 (0,50] (50,100]
##   (0,5]       5        0
##   (5,11]      2        3
```

Możemy wyznaczyć tabele z proporcjami zamiast liczebności.

```
prop.table(tabela)
##         czynnik2
## czynnik1 (0,50] (50,100]
##   (0,5]     0.5      0.0
##   (5,11]    0.2      0.3
```

Wyznaczmy brzegowe liczebności w kolumnach.

```
margin.table(tabela, 2)
## czynnik2
##   (0,50] (50,100]
##        7        3
```

A teraz tabela z liczebnościami brzegowymi.

```
addmargins(tabela)
##         czynnik2
## czynnik1 (0,50] (50,100] Sum
##   (0,5]       5        0   5
##   (5,11]      2        3   5
##   Sum         7        3  10
```

Do generowania wektorów wartości typu czynnikowego, wykorzystać można funkcję `gl()`. Kolejne argumenty tej funkcji opisują liczbę poziomów, liczbę powtórzeń każdego poziomu, długość całego wektora oraz nazwy poziomów. Możemy również określić, czy poziomy są w relacji porządku czy nie. Poniżej przedstawiamy przykład użycia tej funkcji.

```
gl(2, 3, labels = c("Zdrowy", "Chory"))
## [1] Zdrowy Zdrowy Zdrowy Chory  Chory  Chory
## Levels: Zdrowy Chory
```

Tym razem generujemy wektor długości 8 elementów, o dwóch poziomach występujących w dwuelementowych blokach, poziomy są uporządkowane.

```
gl(2, 2, length=8, labels = c("Zdrowy", "Chory"), ordered=TRUE)
## [1] Zdrowy Zdrowy Chory  Chory  Zdrowy Zdrowy Chory  Chory
## Levels: Zdrowy < Chory
```

### 2.1.2 Typ znakowy

Typ znakowy służy do przechowywania napisów (łańcuchów znaków). W tabeli 2.1 zamieszczamy listę najpopularniejszych funkcji do operacji na napisach.

Do prezentacji tych funkcji potrzebujemy zmiennej typu znakowego.

```
nazwa <- "Wroclaw,"
```

Funkcja `paste()` skleja obiekty różnych typów w jeden napis.

```
(napis <- paste(nazwa,"the meeting place\n",sep=" \n"))
## [1] "Wroclaw, \nthe meeting place\n"
```

Funkcją `cat()` wyświetlamy ten napis w sposób niesformatowany.

```
cat(napis)
## Wroclaw,
## the meeting place
```

Funkcją `substr()` wycinamy fragment napisu.

```
substr(napis, 10, 27)
## [1] "\nthe meeting place"
```

Funkcją `strsplit()` rozbijamy napis (tutaj wektor dwóch napisów) na listę wektorów napisów rozdzielanych określonym wzorcem.

```
(tmp <- strsplit(c(nazwa,napis)," "))
## [[1]]
## [1] "Wroclaw,"
## [[2]]
## [1] "Wroclaw," "\nthe"    "meeting"  "place\n"
```

Wyciągnijmy pierwszy element z drugiej listy.

```
tmp[[2]][1]
## [1] "Wroclaw,"
```

Funkcją `print()` wyświetlamy ten napis w sposób sformatowany. Zobaczmy też jak zadziałają funkcje `nchar()` i `toupper()`.

**Tabela 2.1:** Funkcje z pakietu base do pracy na łańcuchach znaków

| | |
|---|---|
| `nchar(wektor)` | Zlicza liczbę znaków w każdym elemencie wektora napisów. Wynikiem jest wektor liczb. |
| `toupper(wektor)` | Zamienia wszystkie litery na duże. |
| `tolower(wektor)` | Zamienia wszystkie litery na małe. |
| `chartr(stare, nowe, wektor)` | Transliteracja. Funkcja zamienia znaki wymienione w argumencie `stare` na inne wymienione w argumencie `nowe` w każdym elemencie wektora. |
| `substr(x, start, stop)` | Wynikiem tej funkcji jest podłańcuch łańcucha znaków `x`, począwszy od znaku o indeksie `start` do znaku o indeksie `stop`. |
| `strsplit(x, wzorzec)` | Ta funkcja rozbija łańcuch znaków na podłańcuchy rozdzielane zadanym wzorcem. W tej funkcji i pozostałych z tej tabeli, opisując wzorzec można używać wyrażeń regularnych. Wynikiem jest lista podłańcuchów. |
| `agrep(wzorzec, x)` | Wynikiem tej funkcji są indeksy elementów wektora, w których w przybliżeniu wystąpił dany wzorzec. W przybliżeniu oznacza, że szukane są dopasowania o odległości edytorskiej Levenshteina, nie większej niż zadany próg (domyślnie, nie różniące się więcej niż jednym znakiem). |
| `sub/gsub(wzorzec, zamiennik, wektor)` | Zamienia pierwsze wystąpienie `sub()` lub wszystkie wystąpienia `gsub()` łańcucha `wzorzec` na `zamiennik` w każdym elemencie wektora. |
| `grep/grepl(wzorzec, wektor)` | Wynikiem tej funkcji są indeksy `grep()` lub wektor wartości logicznych `grepl()`, określający w których elementach wektora wystąpił `wzorzec`. |
| `regexpr(wzorzec, wektor)`, `gregexpr(wzorzec, wektor)`, `regexec(wzorzec, wektor)` | Inne funkcje pozwalające na wyszukiwanie wzorca opisanego przez wyrażenie regularne. |

```
print(napis)
## [1] "Wroclaw, \nthe meeting place\n"
nchar(napis)
## [1] 27
toupper(napis)
## [1] "WROCLAW, \nTHE MEETING PLACE\n"
```

### 2.1.3 Wektory

Wektor, to ciąg obiektów tego samego typu. Może to być ciąg liczb, znaków, łańcuchów znaków lub wartości logicznych. Wektory tworzyć można na wiele sposobów, kilka już poznaliśmy w rozdziale 1.5.5.6.

Używając funkcji `seq()` lub operatora `:` możemy tworzyć sekwencje liczb. Używając funkcji `c()` możemy łączyć obiekty lub wektory w większe wektory. Używając funkcji `vector()`, możemy tworzyć wektor elementów danego typu o ustalonym rozmiarze. Do generowania wektorów określonego typu wykorzystać moż-

na również funkcje `double()`, `integer()`, `character()` oraz `logical()` tworzące wektor elementów danego typu o określonej długości.

Poniżej przedstawiamy przykład tworzenia wektorów różnych typów. Na początek zaicjujemy wektor liczb, napisów i wartości logicznych.

```
(wektor <- vector("integer",10))
##  [1] 0 0 0 0 0 0 0 0 0 0
character(10)
##  [1] "" "" "" "" "" "" "" "" "" ""
logical(10)
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##
```

Długość wektora można wyliczyć funkcją `length()`. Dostęp do elementów wektora uzyskujemy z użyciem operatora `[indeksy]`. W tym przypadku wektor `indeksy` może być w jednej z następujących postaci:

- wektor wartości dodatnich, w tym przypadku z wektora wybrane będą wartości o wskazanych indeksach,
- wektor wartości ujemnych, w tym przypadku z wektora wybrane będą wszystkie wartości poza tymi o wskazanych indeksach,
- wektor nazw, w tym przypadku z wektora wybrane będą wartości określonych przez wskazane nazwy,
- wektor wartości logicznych, w tym przypadku z wektora wybrane będą wszystkie wartości dla których wektor indeksów przyjmuje wartość TRUE.

Używając funkcji `rep()` możemy zwielokrotniać wektory. Poniżej trzykrotnie powtarzamy sekwencję liczb 1:5.

```
(wektor <- rep(1:5, times=3))
##  [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

Długość wektora.

```
length(wektor)
## [1] 15
```

Wybierzmy elementy wektora podając ich indeksy.

```
wektor[c(1:3,6)]
## [1] 1 2 3 1
wektor[-(1:8)]
## [1] 4 5 1 2 3 4 5
```

Wybierzmy elementy wektora mniejsze niż 3. W tym celu zastosujemy indeksowanie wektorem wartości logicznych.

```
wektor[wektor < 3]
## [1] 1 2 1 2 1 2
```

Inne sposoby na powielenie wektora.

```
rep(1:5, length.out=13)
##  [1] 1 2 3 4 5 1 2 3 4 5 1 2 3
rep(1:5, each=3)
##  [1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
```

Jeżeli w wektorze występują wartości powtarzające się, to często będziemy zainteresowani zliczeniem, ile razy poszczególne wartości występują w wektorze. W tym celu można użyć funkcji `table()`, wyznaczającej tablicę kontyngencji lub funkcji `rle()`, kompresującej informację o powtórzonych elementach wektora. Funkcją odwrotną do `rle()`, dekompresującą ww. obiekt jest `inverse.rle()`).

Poniżej przedstawiamy przykłady zastosowań wymienionych funkcji. Inicjujemy przykładowy wektor z danymi i wyznaczamy tabele liczebności.

```
wek <- c(1,2,2,6,6,6,6,6,1,2,2,2,2,1,2,2,3,3,3,4,46,6,6)
table(wek)
## wek
##  1  2  3  4  6 46
##  3  8  3  1  7  1
```

Zapisuje wektor jako pary: wartość, liczba wystąpień.

```
(spakWek <- rle(wek))
## Run Length Encoding
##   lengths: int [1:10] 1 2 1 4 1 2 3 1 1 7
##   values : num [1:10] 1 2 1 2 1 2 3 4 46 6
```

Odtwarzamy wartości w oryginalnym wektorze, została zmieniona oryginalna kolejność.

```
inverse.rle(spakWek)
##  [1] 1 2 2 1 2 2 2 2 1 2 2 3 3 3 4 46 6 6 6 6 6 6 6
```

Kolejność elementów wektora można zmieniać, np. funkcjami `sort()` i `rev()`. Pierwsza porządkuje elementy wektora w kolejności rosnącej (domyślnie) lub malejącej, gdy argument `decreasing=TRUE`. Funkcja `rev()` ustawia elementy w odwrotnej kolejności.

```
(wektor <- c(1:3,0.5 + 1:3, 3))
## [1] 1.0 2.0 3.0 1.5 2.5 3.5 3.0
```

Uporządkujmy elementy wektora.

```
sort(wektor)
## [1] 1.0 1.5 2.0 2.5 3.0 3.0 3.5
```

A teraz malejąco.

```
sort(wektor, decreasing = TRUE)
## [1] 3.5 3.0 3.0 2.5 2.0 1.5 1.0
```

Jeżeli chcemy pozostawić wartości brakujące.

```
sort(wektor, na.last = TRUE)
## [1] 1.0 1.5 2.0 2.5 3.0 3.0 3.5
```

Argument na.last w funkcji `sort()` wskazuje, że jeżeli w wektorze wystąpiły wartości `NA`, to w posortowanym wektorze powinny zostać przeniesione na koniec wektora.

Odwróćmy kolejność elementów w wektorze.

```
rev(wektor)
## [1] 3.0 3.5 2.5 1.5 3.0 2.0 1.0
```

**Tabela 2.2:** Wybrane funkcje z pakietu base do operacji arytmetycznych na wektorze

| | |
|---|---|
| `sum()` | Suma elementów wektora. |
| `diff()` | Różnice kolejnych par elementów wektora. |
| `cumsum()` | Suma skumulowana (ang. *cumulative sum*). Wynikiem tej funkcji jest wektor, w którym na pozycji `i` ma sumę elementów o indeksach `1..i` z wektora wejściowego. |
| `prod()` | Iloczyn elementów wektora. |
| `cumprod()` | Iloczyn skumulowany, tak jak w przypadku `cumsum()` tyle, że zamiast sumy liczone są iloczyny. |
| `min()` | Wartość minimalna w wektorze. |
| `cummin()` | Skumulowana wartość minimalna, czyli wektor wartości minimalnych na podzbiorach `1..i` wektora wejściowego. |
| `max()` | Wartość maksymalna w wektorze. |
| `cummax()` | Skumulowana wartość maksymalna, czyli wektor wartości maksymalnych na podzbiorach `1..i` wektora wejściowego. |
| `pmin()` | Wektor wartości minimalnych. Ta funkcja za argumenty przyjmuje dwa lub więcej wektorów lub macierz. Wynikiem jest wektor wartości, gdzie na pozycji `i` znajduje się wartość minimalna z *i*tego wiersza macierzy lub z *i*tych pozycji wejściowych wektorów. |
| `pmax()` | Wektor wartości maksymalnych. Działanie podobne do funkcji `pmin()` tyle, że liczone są wartości maksymalne. |

Dokonajmy losowej permutacji elementów wektora. Możemy do tego wykorzystać funkcję `sample()` losującą podzbiór bez zwracania.

```
sample(wektor, length(wektor), FALSE)
## [1] 2.0 3.0 2.5 1.0 3.0 3.5 1.5
```

Inne funkcje związane z kolejnością elementów wektora to `rank()` i `order()`. Ich wynikami są, odpowiednio, wektor rang odpowiadających elementom wektora i permutacja, która zmieni kolejność elementów wektora tak, by był on uporządkowany rosnąco. Innymi słowy `rank()` jest odwrotnością funkcji `order()`.

Przykład użycia obu funkcji przedstawiamy poniżej. Wyznaczamy rangi elementów wektora.

```
rank(wektor)
## [1] 1.0 3.0 5.5 2.0 4.0 7.0 5.5
```

Indeks obserwacji w uporządkowanym wektorze.

```
order(wektor)
## [1] 1 4 2 5 3 7 6
```

datayoda: Bing is my friend...I found the cumsum() function. Dirk Eddelbuettel: If bing is your friend, then rseek.org is bound to be your uncle.

datayoda and Dirk Eddelbuettel stackoverflow.com (October 2010)

Jeżeli chcemy wyeliminować powtarzające się elementy w wektorze, to możemy wykorzystać funkcję `unique()`. Jej wynikiem jest wektor wszystkich elementów argumentu wejściowego pomijając powtórzenia. Tę funkcję można również wykorzystać do wyznaczania podzbiorów macierzy lub ramek danych zawierających niepowtarzające się wiersze lub kolumny. Odwrotne działanie ma funkcja `duplicated()`, której wynikiem są indeksy powtarzających się elementów.

### 2.1.4 Listy

Podobnie jak wektory, listy są ciągami obiektów. Pierwsza różnica pomiędzy listami a wektorami polega na możliwych typach elementów składowych. W listach każdy element może być innego typu. Druga różnica, to możliwość odwoływania się do elementów listy za pomocą nazwy tego elementu i operatora $. Elementy wektora mogą posiadać nazwy, nie jest jednak możliwe odwoływanie się do tych elementów z użyciem operatora $.

Poniżej tworzymy listę złożoną z czterech elementów, wektora, dwóch napisów i wartości logicznej. Odwołujemy się przez nazwę do elementu drugiego.

```
lista <- list(1:3, imie="Bartek", nazwisko="Nowak", czyKobieta=FALSE)
lista$imie
## [1] "Bartek"
```

Odwołujemy się do pierwszego elementu listy operatorem [], wynikiem jest jednoelementowa lista zawierająca wektor.

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

Do elementów list powinniśmy odwoływać się operatorem [[]].

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

Pierwszy element listy to wektor, możemy więc użyć takiego zapisu.

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

Funkcja `lapply()` pozwala na wykonanie określonego działania (tutaj zamiana liter na duże), na każdym z elementów listy.

```
lapply(lista,toupper)
## [[1]]
## [1] "1" "2" "3"
## $imie
## [1] "BARTEK"
## $nazwisko
## [1] "NOWAK"
## $czyKobieta
## [1] "FALSE"
```

### 2.1.5 Ramki danych

Ramka danych (ang. *data frame*) to struktura do przechowywania danych tabelarycznych. Korzystać z tego obiektu można tak jak z listy, której każdy element jest wektorem, a więc z użyciem operatora `$` (wektory odpowiadają kolejnym kolumnom) lub jak z macierzy dwuwymiarowej (korzystając z operatora `[,]`). W przeciwieństwie do macierzy różne kolumny mogą mieć elementy różnych typów.

Aby obejrzeć elementy ramki danych można użyć funkcję `edit()`. Otwiera ona okno, w którym można przeglądać wartości w ramce danych, a także modyfikować

jej poszczególne elementy. Po zakończeniu edycji, jeżeli jakiś element ramki został zmieniony, to wynikiem funkcji `edit()` jest zmieniona ramka danych. Podobnie działa funkcja `fix()`, z tą różnicą, że nie zwraca ona wyniku, ale automatycznie zmienia wartość ramki danych będącej argumentem tej funkcji. Edytować elementy ramki danych możemy również funkcją `data.entry()`.

Innym sposobem wyświetlenia fragmentu ramki danych jest skorzystanie z funkcji `head()` i `tail()`. Wyświetlają one kilka (liczbę określoną przez drugi argument, domyślnie sześć) pierwszych lub ostatnich wierszy z ramki danych/macierzy.

Wynikiem funkcji `names()` i `colnames()` jest wektor z nazwami kolumn (zmiennych) w ramce danych. Te funkcje pozwalają również na zmianę tych nazw. Podobnie funkcja `rownames()` pozwala na wyświetlenie lub zmianę nazwy wierszy (przypadków/obserwacji) w ramce danych. Oba wektory (z nazwami kolumn i wierszy) można otrzymać w wyniku wywołania funkcji `dimnames()`. Poprawne wektory nazw nie mogą zawierać powtarzających się wartości.

Liczbę wierszy i kolumn, tak jak dla macierzy jak i ramki danych, można wyznaczyć funkcją `dim()`, (jej wynikiem jest dwuelementowy wektor z wymiarami ramki danych) lub funkcjami `ncol()` i `nrow()` wyznaczającymi odpowiednio liczbę kolumn i liczbę wierszy.

Jeżeli planujemy wykonać wiele operacji na kolumnach ramki danych, to przydać mogą się funkcje `attach()` i `detach()`. Pierwsza z nich powoduje, że kolumny ramki danych są widoczne w przestrzeni nazw jako wektory o nazwach odpowiadających nazwom kolumn. A więc, aby odwołać się do kolumny, nie potrzebujemy podawać nazwy ramki danych, tym samym możemy skrócić zapis wielu poleceń. Druga funkcja usuwa z przestrzeni nazw te obiekty.

Używając funkcji `attach()` należy pamiętać o usunięciu nazw kolumn gdy już nie będą potrzebne, ponieważ duża liczba obiektów w przestrzeni nazw może prowadzić do kolizji oznaczeń i trudnych do wykrycia błędów.

Przedstawmy obie wymienione funkcje na przykładzie. Funkcją `data()` wczytujemy ramkę danych z danymi o irysach. Jakie są nazwy kolumn w tej ramce?

```
data(iris)
names(iris)
## [1] "Sepal.Length" "Sepal.Width" "Petal.Lengt" "Petal.Widt" "Species"
colnames(iris)
## [1] "Sepal.Length" "Sepal.Width" "Petal.Lengt" "Petal.Widt" "Species"
```

Wyświetlmy kilka pierwszych wartości ostatniej kolumny.

```
head(iris$Species)
## [1] setosa setosa setosa setosa setosa setosa
## Levels: setosa versicolor virginica
```

Po użyciu funkcji `attach()` możemy posługiwać się nazwami kolumn ramki danych `iris` bez potrzeby poprzedzania ich przedrostkiem `iris$`.

```
attach(iris)
head(Species)
## [1] setosa setosa setosa setosa setosa setosa
## Levels: setosa versicolor virginica
```

Nie zawsze chcemy zaśmiecać całą przestrzeń roboczą nazwami wszystkich kolumn z tabeli, tylko po to, by skrócić zapis nazw zmiennych w jednej linii. W wielu przypadkach wygodniej jest użyć funkcji `with()`, której pierwszy argument wskazuje ramkę danych a drugi zbiór poleceń. W tych poleceniach można odwoływać się do nazw kolumn bez konieczności wskazywania ramki danych.

Operator <<- jest opisany w rozdziale 1.6.2.4. W tym przykładzie zastępuje operator <- aby zapisać wynik w globalnej przestrzeni nazw a nie w przestrzeni nazw ramki iris.

Poniżej używamy funkcji `with()`, aby skrócić zapis, w tym przypadku wewnątrz tej funkcji możemy korzystać bezpośrednio z kolumn ramki danych `iris`.

```
with(iris, roznica <<- Sepal.Length - Petal.Length)
range(roznica)
## [1] 0.3 4.6
dane <- list(od = 10, do = 20, kiedy = "rano")
with(dane, roznica <<- do - od)
roznica
## [1] 10
```

Jeżeli chcemy wykonać obliczenia jedynie na wybranych wierszach z ramki danych, a nie na całej ramce, to do wyboru wierszy można użyć funkcji `subset()`. Pierwszym argumentem jest ramka danych z której chcemy wybrać wiersze a drugim wektor indeksów lub wektor wartości logicznych określających, które wiersze mają być wybrane. Inną funkcją o podobnym działaniu jest funkcja `split()`. Pierwszym argumentem tej funkcji jest ramka danych, a drugim wektor wartości typu wyliczeniowego/czynnikowego o długości równej liczbie wierszy w ramce danych. Wynikiem działania tej funkcji jest lista ramek danych, której kolejne elementy zawierają ramki danych z wierszami odpowiadającymi tej samej wartości zmiennej grupującej.

Do konwersji ramek danych zawierających zmienne czynnikowe z postaci wąskiej na szeroką, przydają się też funkcje `unstack()` i `stack()`. Jeżeli mamy zebrane pomiary pewnej cechy w różnych podpopulacjach, to możemy wyniki tych pomiarów przedstawić albo w tabeli, w której w pierwszej kolumnie jest informacja o podpopulacji, a w drugiej informacja o pomiarze albo w tabeli, w której w kolejnych kolumnach są zebrane informacje o wynikach dla podpopulacji. Funkcje `unstack()` i `stack()` pozwalają na konwertowanie danych z jednej postaci na drugą. Zobaczmy przykłady dla wybranych funkcji.

```
czynniki <- factor(rep(c("a", "b", "c"), times=5))
dane     <- data.frame(runif(15), czynniki)
dim(dane)
##  [1] 15  2
```

Wyświetlamy pierwsze sześć wierszy tej ramki danych.

```
head(dane)
##    runif.15. czynniki
## 1 0.72091831        a
## 2 0.03992616        b
## 3 0.27803642        c
## 4 0.08969056        a
## 5 0.24805597        b
## 6 0.15816209        c
```

Zamieniamy ramkę na macierz o bardziej zawartym formacie. Wartości odpowiadające tym samym czynnikom są w tej samej kolumnie.

```
unstack(dane)
##            a          b         c
## 1 0.72091831 0.03992616 0.2780364
## 2 0.08969056 0.24805597 0.1581621
## 3 0.44452236 0.41022322 0.6236955
## 4 0.02504580 0.99141448 0.3911667
## 5 0.13058592 0.60403541 0.5152448
```

Funkcja split() podzieli tę ramkę na trzy ramki.

```
length(split(dane, czynniki))
## [1]  3

split(dane, czynniki)[[1]]
##     runif.15. czynniki
## 1  0.72091831        a
## 4  0.08969056        a
## 7  0.44452236        a
## 10 0.02504580        a
## 13 0.13058592        a
```

Wybieramy podzbiór danych dla czynnika a.

```
subset(dane, czynniki=="a")
##     runif.15. czynniki
## 1  0.72091831        a
## 4  0.08969056        a
## 7  0.44452236        a
## 10 0.02504580        a
## 13 0.13058592        a
```

### 2.1.6 Macierze

Macierz to struktura do przechowywania wartości tego samego typu w postaci tabelarycznej. Dostęp do elementów macierzy odbywa się podobnie jak dla wektorów, poprzez wskazanie indeksów wybranych elementów korzystając z operatora [indeksyWierszy,indeksyKolumn].

Indeksy można podać dla każdego wymiaru oddzielnie, jeżeli indeksy nie będą podane to zostanie wybrana cała kolumna/wiersz. Format podawanych indeksów jest taki jak dla wektorów, czyli może to być wektor liczb albo dodatnich, albo ujemnych albo wektor nazw albo wektor wartości logicznych.

Firstly, don't call your matrix 'matrix'. Would you call your dog 'dog'? Anyway, it might clash with the function 'matrix'.

Barry Rowlingson, R-help (October 2004) fortunes(77)

Poniższy przykład przedstawia różne sposoby indeksowania. Zacznijmy od skonstruowania przykładowej macierzy o wymiarach 3x3.

```
(macierz <- matrix(1:9,3,3))
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

Wektor na diagonali tej macierzy.

```
diag(macierz)
## [1] 1 5 9
```

Wartość elementów z dwóch pierwszych kolumn i wierszy zamieniamy na 1.

```
(macierz[1:2,1:2] <- 1)
##      [,1] [,2] [,3]
## [1,]    1    1    7
## [2,]    1    1    8
## [3,]    3    6    9
```

Wartości elementów w dwóch pierwszych wierszach zamieniamy na 2.

```
(macierz[1:2,] <- 2)
##      [,1] [,2] [,3]
## [1,]    2    2    2
## [2,]    2    2    2
## [3,]    3    6    9
```

Wszystkie elementy macierzy zamieniamy na 3, ale wymiar się nie zmienia.

```
(macierz[] <- 3)
##      [,1] [,2] [,3]
## [1,]    3    3    3
## [2,]    3    3    3
## [3,]    3    3    3
```

Ta instrukcja przypisuje do zmiennej macierz liczbę 4, ta zmienna już nie przechowuje macierzy.

```
(macierz <- 4)
## [1] 4
```

Wyznaczmy macierz odwrotną.

```
macierzOdwrotna <- solve(macierz)
```

W ostatnim wierszu tego przykładu została wykorzystana funkcja `solve()`. Działanie tej funkcji zależy od liczby argumentów. Jeżeli podany jest tylko jeden argument, który jest macierzą kwadratową, to funkcja `solve()` wyznacza macierz odwrotną do danej macierzy. Jeżeli podane są dwa argumenty to funkcja `solve()` wyznacza rozwiązanie układu równań liniowych postaci

$$Ax = B,$$

gdzie $A$ to macierz współczynników, $B$ to kolumnowy wektor, a $x$ to wektor szukanych wartości. Aby wyznaczyć wektor $x$ należy wywołać funkcję `solve(A, B)`.

W powyższym przykładzie użyliśmy również funkcji `diag()`. Działanie tej funkcji zależy od tego co jest wartością pierwszego argumentu. Jeżeli pierwszym argumentem tej funkcji jest macierz, to wynikiem będzie wektor elementów na przekątnej. Jeżeli argumentem jest wektor, to wynikiem funkcji `diag()` jest macierz diagonalna z elementami wektora argumentu na przekątnej i zerami poza. Jeżeli argumentem będzie jedna liczba całkowita, to wynikiem będzie macierz identycznościowa o liczbie wierszy i kolumn określonych przez ww. argument.

W przykładzie na poprzedniej stronie, przypisując wartości do wiersza, kolumny, fragmentów macierzy jako „wypełnienie" podawaliśmy jedną wartość. Przy przypisywaniu obowiązuje tzw. zasada replikacji, ang. *recycling rule*. Najprościej rzecz ujmując, aby wypełnić $x$ pól macierzy lub wektora możemy podać $y$ wartości, pod warunkiem, że $x$ jest wielokrotnością $y$. Przykładowo, jeżeli chcemy zmienić sześć pól w wektorze lub macierzy, to możemy podać po prawej stronie operatora przypisania wektor sześcioelementowy, trójelementowy, dwu- lub jednoelementowy. Krótszy wektor zostanie zwielokrotniony tyle razy ile potrzeba, by wypełnić wszystkie wskazane pola. Jeżeli dłuższy wektor nie jest wielokrotnością krótszego, to zasada replikacji też może zadziałać, jednak zgłoszony zostanie błąd lub ostrzeżenie.
Poniżej przedstawiamy kilka przykładowych zastosowań tej reguły.

Recycling rule działa! dodajemy wektor 10-elementowy do 2-elementowego.

```
(1:10) + (1:2)
## [1]  2  4  4  6  6  8  8 10 10 12
```

A teraz kilka przykładów na macierzach, wypełnijmy całą macierz.

```
macierz <- matrix(0,2,3)
(macierz[] <- 1:6)
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

Krótsze wektory zostaną zwielokrotnione.

```
(macierz[] <- 1:3)
##      [,1] [,2] [,3]
## [1,]    1    3    2
## [2,]    2    1    3

(macierz[] <- 1:2)
##      [,1] [,2] [,3]
## [1,]    1    1    1
## [2,]    2    2    2

(macierz[] <- 1)
##      [,1] [,2] [,3]
## [1,]    1    1    1
## [2,]    1    1    1
```

W poprzednich przykładach pokazaliśmy jak wykonywać na macierzach dwa rodzaje iloczynów: iloczyn macierzowy i iloczyn element z elementem, służyły do tego celu operatory `%*%` i `*`. Innym iloczynem, który możemy chcieć wykonać jest iloczyn Kroneckera wykonywany przez funkcję `kronecker()` oraz `outer()`.

Funkcja `outer()` ma większe możliwości, może być wykorzystana np. do generowania siatek argumentów. Polecenie `outer(obi1, obi2, fun)` spowoduje wykonanie funkcji `fun` dla każdej kombinacji elementów zmiennej `obi1` z elementami zmiennej `obi2` (zmienne `obi1` i `obi2` mogą być wektorami lub macierzami).

Poniżej przedstawimy kilka przykładów użycia. Elementy wynikowej macierzy to iloczyny mieszane argumentów.

```
outer(1:4,1:4)
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    2    4    6    8
## [3,]    3    6    9   12
## [4,]    4    8   12   16
```

Elementy wynikowej macierz to wartości z pierwszego argumentu.

```
outer(1:4,1:4, function(x,y) x)
##      [,1] [,2] [,3] [,4]
## [1,]    1    1    1    1
## [2,]    2    2    2    2
## [3,]    3    3    3    3
## [4,]    4    4    4    4
```

Poniżej tworzymy napisy składające się z litery, znaku * i cyfry

```
outer(letters[1:4], 1:5, paste, sep="*")
##      [,1]  [,2]  [,3]  [,4]  [,5]
## [1,] "a*1" "a*2" "a*3" "a*4" "a*5"
## [2,] "b*1" "b*2" "b*3" "b*4" "b*5"
## [3,] "c*1" "c*2" "c*3" "c*4" "c*5"
## [4,] "d*1" "d*2" "d*3" "d*4" "d*5"
```

Możemy policzyć funkcję na kombinacjach macierzy i wektora, wynikiem będzie trójwymiarowa macierz.

```
mat1 <- matrix(LETTERS[1:4],2,2)
mat2 <- c(1:3)
outer(mat2, mat1, paste)
## , , 1
##      [,1]  [,2]
## [1,] "1 A" "1 B"
## [2,] "2 A" "2 B"
## [3,] "3 A" "3 B"
##
## , , 2
##      [,1]  [,2]
## [1,] "1 C" "1 D"
## [2,] "2 C" "2 D"
## [3,] "3 C" "3 D"
```

Funkcję `outer()` można też wykorzystać do wygenerowania wszystkich kombinacji poziomów dla zmiennych czynnikowych. Ten same efekt można uzyskać też funkcją `expand.grid()`. Wynikiem jest macierz, której wiersze zawierają wszystkie kombinacje zmiennych wskazanych jako argumenty tej funkcji.

Jak? To pytanie czytelnik może potraktować jako zadanie domowe.

```
expand.grid(edukacja= c("uczen", "absolwent"),
        praca = c("pracuje", "nie pracuje"),
        plec = c("Kobieta", "Mezczyzna"))
##    edukacja       praca      plec
## 1     uczen     pracuje   Kobieta
## 2 absolwent     pracuje   Kobieta
## 3     uczen nie pracuje   Kobieta
```

```
## 4 absolwent nie pracuje   Kobieta
## 5     uczen     pracuje Mezczyzna
## 6 absolwent     pracuje Mezczyzna
## 7     uczen nie pracuje Mezczyzna
## 8 absolwent nie pracuje Mezczyzna
```

Aby uporządkować wiersze macierzy według określonego klucza (wybranej kolumny) możemy użyć funkcji `order()`, omówionej przy okazji opisywania wektorów. Wynikiem tej funkcji jest permutacja, która powoduje że elementy wektora będą uporządkowane. Tak otrzymaną permutację można wykorzystać do posortowania macierzy lub ramki danej względem wybranej kolumny.
Zainicjujmy przykładową macierz 16 liczb.

```
(macierz <- matrix(sample(1:16),4,4, byrow=T))
##      [,1] [,2] [,3] [,4]
## [1,]   14    6   16    8
## [2,]   12   11    3    7
## [3,]   10    2    5   15
## [4,]    4    1   13    9
```

Sortujemy macierz względem pierwszej kolumny, funkcją order() odczytujemy pożądaną kolejność i przestawimy wiersze macierzy.

```
macierz[order(macierz[,1]), ]
##      [,1] [,2] [,3] [,4]
## [1,]    4    1   13    9
## [2,]   10    2    5   15
## [3,]   12   11    3    7
## [4,]   14    6   16    8
```

Przydatną funkcją do operacji na macierzach jest funkcja `apply()`, która pozwala na wykonanie pewnej operacji na kolumnach, wierszach lub wartościach zadanej macierzy. Dzięki temu można policzyć sumy, średnie, maksima lub inne funkcje z kolumn, czy też wierszy macierzy. Przedstawimy to na przykładach.
Na tej macierzy będziemy eksperymentować.

```
(macierz <- matrix(1:16,4,4))
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
## [3,]    3    7   11   15
## [4,]    4    8   12   16
```

Wyznaczmy sumę elementów w wierszach.

```
apply(macierz,1,sum)
## [1] 28 32 36 40
```

A teraz sumę elementów w kolumnach.

```
apply(macierz,2,sum)
## [1] 10 26 42 58
```

Elementy maksymalne w wierszach.

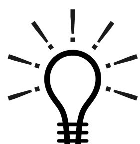
```
apply(macierz,1,max)
## [1] 13 14 15 16
```

A teraz sumę kwadratów elementów w wierszach.

```
apply(macierz,1,function(x) sum(x^2))
## [1] 276 336 404 480
```

I operacja na każdym elemencie macierzy. Tutaj reszta z dzielenia przez 5.

```
apply(macierz,c(1,2),function(x) {x %% 5})
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    4    3
## [2,]    2    1    0    4
## [3,]    3    2    1    0
## [4,]    4    3    2    1
```



Wyznaczanie średnich i sum z kolumn lub wierszy macierzy w funkcją `apply()` należy traktować jedynie jako przykład. Znacznie szybciej można to zrobić funkcjami `colSums()`, `rowSums()`, `colMeans()` i `rowMeans()`.
Pomijając ten przypadek, funkcja `apply()` jest bardzo przydatna do wyznaczania różnych charakterystyk macierzy lub ramek danych, jest też znacznie szybsza niż odpowiadająca jej pętla for.

Macierze można też tworzyć funkcjami `cbind()` i `rbind()`. Funkcja `cbind()` pozwala na zbudowanie macierzy poprzez połączenie ze sobą kilku wektorów (wektory będą kolumnami w powstałej macierzy), macierzy lub przez dodanie kolumny do już istniejącej macierzy. Wynikiem tej funkcji jest macierz powstała z połączenia kolumn wszystkich argumentów (macierzy lub wektorów). Podobnie funkcja `rbind()` tworzy macierz powstałą z połączenia wierszy wszystkich argumentów. W obu przypadkach, w razie potrzeby, argumenty są konwertowane do wspólnego typu. Obie funkcje można traktować jak macierzowe odpowiedniki funkcji `c()`, która służyła do łączenia wektorów.

```
rbind(1:4,"a")
##      [,1] [,2] [,3] [,4]
## [1,] "1"  "2"  "3"  "4"
## [2,] "a"  "a"  "a"  "a"
```

Łączyć macierze ze sobą można również funkcją `merge()`. Działa ona podobnie do SQLowego polecenia JOIN. Dla dwóch macierzy lub ramek danych będących argumentami tej funkcji możemy wskazać kolumny (jedną lub więcej) z kluczami (wskazujące na identyczne obiekty). Funkcja `merge()` tworzy z dwóch macierzy jedną, łącząc wiersze o takiej samej wartości klucza we wskazanych kolumnach.

Na potrzeby przykładu skonstruujmy dwie ramki danych ze wspólną kolumną `klasa`.

```
(tab1 <- data.frame(uczen = c("Ala", "Ola", "Ela", "Ula"),
            klasa = c("IA", "IB", "IB", "IB")))
##   uczen klasa
## 1   Ala    IA
## 2   Ola    IB
## 3   Ela    IB
## 4   Ula    IB
```

Ramka do złączenia z ramką `tab1`.

```
(tab2 <- data.frame(klasa = c("IA", "IB"), wychowawca = c("J Kowalski",
    "B Nowak"), profil = c("mat-fiz", "bio-chem")))
##   klasa wychowawca   profil
## 1    IA J Kowalski  mat-fiz
## 2    IB    B Nowak bio-chem
```

Sklejamy obie ramki łącząc obiekty o identycznych wartościach kolumny klasa.

```
merge(tab1, tab2)
##   klasa uczen wychowawca   profil
## 1    IA   Ala J Kowalski  mat-fiz
## 2    IB   Ola    B Nowak bio-chem
## 3    IB   Ela    B Nowak bio-chem
## 4    IB   Ula    B Nowak bio-chem
```

Dotąd pracowaliśmy na dwuwymiarowych macierzach, te są najczęściej spotykane w algebrze, ale wymiar macierzy w programie R nie jest ograniczony do dwóch. Możemy tworzyć i korzystać z macierzy o praktycznie dowolnym wymiarze. Jedynym problemem dla dużych macierzy jest trudność ich wypisywania na ekranie. Poniżej przedstawiamy przykład tworzenia i operacji na macierzy o 4 wymiarach. Na takiej macierzy można również operować funkcją `apply()`.

Poniżej wypełniamy losowymi wartościami macierz o wymiarach 3x3x3x3.

```
macierz <- matrix(runif(3^4))
dim(macierz) <- c(3,3,3,3)
```

Zobaczmy teraz dwuwymiarowe przecięcie tej macierzy.

```
macierz[1,1,,]
##           [,1]      [,2]      [,3]
## [1,] 0.5421688 0.9960112 0.8526290
## [2,] 0.6184002 0.4275497 0.4427460
## [3,] 0.9981847 0.7094703 0.8201444
```

Wyznaczmy maksimum elementów w odpowiednich podmacierzach. Ustalmy wartości w trzech wybranych komórkach, by można było sprawdzić czy maksima są dobrze liczone.

```
macierz[1,1,,1] <- c(2,3,4)
apply(macierz,1,max)
## [1] 4.0000000 0.9596400 0.9759735
apply(macierz,c(1,3),max)
##           [,1]      [,2]      [,3]
## [1,] 2.0000000 3.0000000 4.0000000
## [2,] 0.9180873 0.7836242 0.9596400
## [3,] 0.7252060 0.9285664 0.9759735
```

Ta tabela opisuj ile osób zginęło a ile przeżyło w podziale na na płeć, wiek, i klasę w której pasażer podróżował.

Zbiór danych `Titanic` jest czterowymiarową tabelą. Policzmy sumy brzegowe dla wymiaru 2 i 4, czyli dla informacji o klasie w której osoba podróżowała i wiek pasażera.

```
dim(Titanic)
## [1] 4 2 2 2
```

**Tabela 2.3:** Funkcje do operacji na macierzach lub ich wierszach/kolumnach

| | |
|---|---|
| `qr()` | Funkcja do wyznaczania dekompozycji QR macierzy. |
| `eigen()` | Funkcja do wyznaczania wartości własnych i wektorów własnych macierzy. |
| `svd()` | Funkcja do wyznaczania dekompozycji SVD macierzy. |
| `det()` | Funkcja do wyznaczania wyznacznika. |
| `upper.tri()` | Wynikiem funkcji są indeksy elementów znajdujących się powyżej przekątnej macierzy. |
| `lower.tri()` | Wynikiem funkcji są indeksy elementów znajdujących się poniżej przekątnej macierzy. |
| `solve()` | Funkcja do wyznaczania odwrotności macierzy lub rozwiązywania układu równań liniowych. |
| `t()` | Wykonuje transpozycje macierzy, używając języka potocznego, zamienia kolumny z wierszami. |
| `colSums()` | Wylicza sumy elementów w każdej kolumnie. |
| `rowSums()` | Wylicza sumy elementów w każdym wierszu. |
| `colMeans()` | Wylicza średnie elementów w każdej kolumnie. |
| `rowMeans()` | Wylicza średnie elementów w każdym wierszu. |
| `rowsum()` | Jeżeli pierwszy argument tej funkcji jest macierzą o wymiarach $n \times m$ a argument drugi argument jest wektorem $n$ elementowym o wartościach typu czynnikowego o $k$ poziomach, to funkcja `rowsum()` dla każdej kolumny z wejściowej macierzy wyznacza sumy elementów pogrupowanych przez czynniki zmiennej grupującej. Wynikiem jest więc macierz wymiaru $k \times m$. |

Który wymiar czemu odpowiada?

```
dimnames(Titanic)
## $Class
## [1] "1st"  "2nd"  "3rd"  "Crew"
## $Sex
## [1] "Male"   "Female"
## $Age
## [1] "Child" "Adult"
## $Survived
## [1] "No"  "Yes"
```

Sumy brzegowe policzymy funkcją `sum()`. Musimy tylko wskazać odpowiednie podmacierze.

```
apply(Titanic, c(1,3), sum)
##       Age
## Class  Child Adult
##   1st      6   319
##   2nd     24   261
##   3rd     79   627
##   Crew     0   885
```

Macierz o więcej niż dwóch wymiarach jest obiektem klasy `array` a nie `matrix`. Dla użytkownika nie ma różnicy w sposobie korzystania z takich macierzy, dlatego nie rozróżnialiśmy tych dwóch klas w prezentowanych powyżej opisach.
Wielowymiarowe macierze tworzy się albo używając funkcji `dim()` (przykład powyżej) albo używając konstruktora `array()`. Przykładowo wywołanie `array(data, dim)` utworzy macierz o wymiarach opisanych wektorem `dim` zainicjowaną wartościami z argumentu `data`.

### 2.1.7 Obiekty

W tabeli 2.4 przedstawiamy listę funkcji, które można wykonać na obiektach dowolnego typu. Służą one najczęściej do weryfikacji oraz zmiany wybranych właściwości lub treści zadanych obiektów.

Zobaczmy czym różni się typ od klasy na przykładzie klasy factor, której wewnętrzna reprezentacja to liczby całkowite.

```
(imie <- factor("Karolina"))
## [1] Karolina
## Levels: Karolina
class(imie)
## [1] "factor"
mode(imie)
## [1] "numeric"
typeof(imie)
## [1] "integer"
```

Używając funkcji `str()` można sprawdzić strukturę obiektu, czyli jakie pola i o jakich wartościach ma dany obiekt.

```
wynikhist <- hist(1:10, plot=FALSE)
class(wynikhist)
## [1] "histogram"
```

Wynik wywołania funkcji `hist()` to nie tylko wykres, to również obiekt przechowujący informacje o środkach wyznaczonych przedziałów oraz o wysokościach poszczególnych słupków.

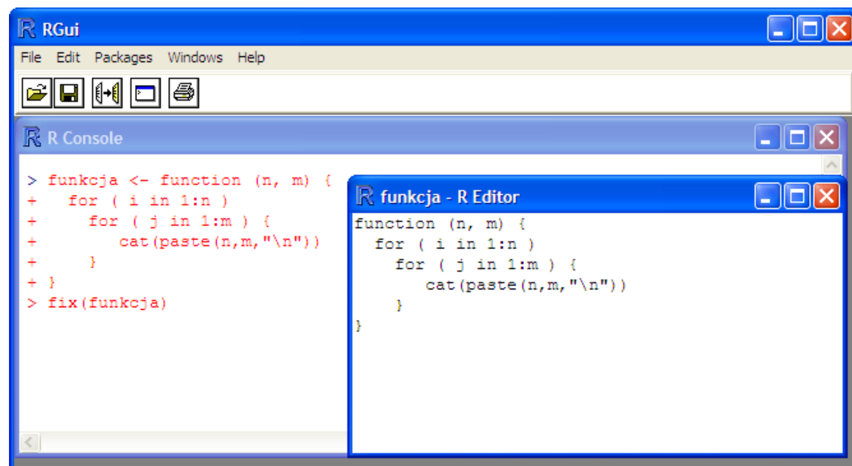
```
str(wynikhist)
## List of 7
##  $ breaks     : num [1:6] 0 2 4 6 8 10
##  $ counts     : int [1:5] 2 2 2 2 2
##  $ intensities: num [1:5] 0.1 0.1 0.1 0.1 0.1
##  $ density    : num [1:5] 0.1 0.1 0.1 0.1 0.1
##  $ mids       : num [1:5] 1 3 5 7 9
##  $ xname      : chr "1:10"
##  $ equidist   : logi TRUE
##  - attr(*, "class")= chr "histogram"
```

Poświęćmy więcej uwagi funkcji `str()` przedstawiającej wewnętrzną strukturę obiektu. Jest to bardzo przydana funkcja gdy w wiemy, że pewna informacja jest przechowywana w obiekcie ale nie wiemy w jakim polu lub gdy nie wiemy jakie in-

formacje w obiekcie się znajdują. Przykładowo wynikiem funkcji `lm()` jest wiele częściowych informacji wyznaczanych podczas estymacji współczynników modelu, używając funkcji `str()` można dowiedzieć się jakie.

**Tabela 2.4:** Funkcje z pakietów `utils` i `base` sprawdzające lub zmieniające właściwości obiektów

| | |
|---|---|
| `fix(obiekt)` | Funkcja otwiera okno edytora i pozwala na zmianę wartości zmiennej `obiekt`. W ten sposób możemy łatwo zmodyfikować wartość zmiennej, funkcji lub ramki danych. Szczególnie przydatne jest to polecenie do modyfikacji wartości funkcji (nie musimy wklejać całego kodu funkcji od początku) oraz ramek danych (modyfikacja wartości wybranych pól jest dużo wygodniejsza). |
| `class(obiekt)` | Wynikiem tej funkcji jest nazwa klasy obiektu. Nazwę klasy obiektu można dowolnie modyfikować nie zmieniając zawartości obiektu. Nawet jednak, jeżeli zawartość jest taka sama, to różne funkcje zastosowane do danego obiektu różniącego się tylko etykietką klasy mogą dać różne wyniki. |
| `unclass(obiekt)` | Wynikiem jest wartość obiektu `obiekt` z usuniętym atrybutem `class`. Usuwane są poprzednie zmiany wykonane funkcją `class()`. |
| `typeof(obiekt)` | Wynikiem tej funkcji jest nazwa typu zmiennej. |
| `mode(obiekt)` | Wynikiem tej funkcji jest informacja o sposobie przechowywania obiektu. W większości przypadków wyniki będą takie jak dla funkcji `typeof()`, z wyjątkiem typów, które mają identyczną reprezentacje wewnętrzną, np. zarówno typ „double” jak i „integer” są przechowywane jako liczba, a wiec mają reprezentacje „numeric”. |
| `attributes(obiekt)` | Wynikiem tej funkcji jest lista atrybutów danego obiektu, używając tej funkcji można również modyfikować wartości atrybutów. |
| `attr(obiekt, atrybut)` | Wynikiem jest wartość wybranego atrybutu danego obiektu. Używając tej funkcji po prawej stronie operatora `<-` można zmienić wartość danego atrybutu. |
| `comment(obiekt)` | Wyświetla lub pozwala na zmianę komentarza dla danego obiektu. Komentarz odpowiada atrybutowi o nazwie `comment` (patrz funkcja `verb()`). Od pozostałych atrybutów atrybut `comment` różni się tym, że nie jest wypisywany na ekranie przez funkcję `print()`. |
| `object.size(obiekt)` | Wynikiem tej funkcji jest liczba bajtów pamięci zajmowanych przez argument. |
| `str(obiekt)` | Wynikiem tej funkcji jest informacja o strukturze danego obiektu, tj. nazwach i rozmiarach poszczególnych atrybutów, zmiennych itp. |




**Rysunek 2.1:** Przykładowe wywołanie funkcji `fix()`. Korzystając z niej możemy łatwo edytować funkcję lub obiekty różnych typów.

Funkcja `object.size()` wyznacza rozmiar wskazanego obiektu w bajtach.

```
object.size(wynikhist)
## [1] 976
```

Zobaczmy jaka jest struktura wewnętrznego opisu list.

```
fix(wynikhist)
```

### 2.1.8 Klasy

Listy będąc agregatami wartości dowolnych typów pełnią w języku R podobną funkcję jak klasy lub rekordy w innych językach programowania. Jeżeli wynikiem działania funkcji ma być zbiór wartości różnych typów, wygodnie jest je przekazać jako listy, której kolejne elementy mogą być wartościami różnych typów. Takiej liście możemy przypisać atrybut `class` i myśleć o niej jako o obiekcie klasy. Taki rodzaj klas nazywany jest klasami S3. Dla takich klas nie można określić żadnego formalnego ograniczenia na zawartość obiektu w zależności od nazwy klasy.

W języku R można też tworzyć złożone obiekty w inny sposób, wzorowany na klasach S4 znanych z języka S. Jest to sposób dostępu do elementów klasy bardziej podobny do takich języków obiektowych jak C++ czy Java.

Różnic pomiędzy klasami S3 i S4 jest wiele. Np. dla klas S3 pomiędzy dwoma listami nie można określić żadnych formalnych relacji, nie można określić też ograniczeń na typy i liczbę elementów list. W przypadku klas S4 takie ograniczenia na wartości składowe obiektów można narzucać.

Zanim opiszemy dokładniej klasy S4 pokażmy krótki przykład w którym użyjemy raz klas S3 raz klas S4 aby zdefiniować klasę i generyczną funkcje wypisującą obiekty danej klasy na ekranie.

Dla klas S3 nie ma potrzeby deklarować reprezentacji obiektów tej klasy. Ale dla klas S4 musimy jawnie określić reprezentację.

```
setClass("osobaS4", representation(imie="character", wiek="numeric"))
```

Dla obiektów klasy S3 funkcje generyczne tworzy się dopisując sufiks określający nazwę klasy, poniżej definiujemy generyczną funkcję dla klasy S3, nie ma pewności że argument x będzie miał wymagane pola ponieważ struktura klasy osobaS3 nie jest nigdzie jawnie określona.

```
print.osobaS3 <- function(x) {
   cat(x$imie, " to osobnik w wieku ", x$wiek, "\n\n")
}
```

Dla klasy S4 funkcje generyczne tworzy się funkcją `setMethod()`.

```
setMethod("print", "osobaS4",
 function(x) {
   cat(x@imie, " to osobnik w wieku ", x@wiek, "\n\n")
})
```

Nowe obiekty klasy S3 tworzy się poprzez zmianę atrybutu class z użyciem funkcji `class()`, nie jest wymagana zgodność z żadnym prototypem. Nowe obiekty klasy S4 tworzy się funkcją new(), pola muszą być zgodne z deklaracją klasy.

```
o1 <- list(imie = "Przemek", wiek = 27)
class(o1) <- "osobaS3"
o2 <- new("osobaS4", imie="Przemek", wiek=27)
```

Wywołujemy funkcję `print()` dla obiektów obu klas.

To już nieaktualne informacje. Ale aż żal zmieniać.

```
o1
## Przemek  to osobnik w wieku  27
print(o2)
## Przemek  to osobnik w wieku  27
```

Nową klasę S4 konstruuje się poleceniem `setClass()`. Tworząc klasę określamy jej nazwę oraz definiujemy jej reprezentacje. Z każdą klasą S4 stowarzyszone są następujące pojęcia:

- Sloty. Sloty to elementy (pola) obiektów klasy. Dostęp do slotów możliwy jest dzięki operatorowi `@` oraz dzięki funkcji `methods::slot()`. W różnych slotach danej klasy przechowywane mogą być wartości różnych typów, jednak w odpowiadających sobie slotach różnych obiektów tej samej klasy muszą być przechowywane obiekty tego samego typu.

  Sloty nie mogą być indeksowane liczbami naturalnymi w przeciwieństwie do elementów list ze slotów można korzystać tylko podając ich nazwę.

- Nadklasy (ang. *superclass*, inne stosowane nazwy: klasa nadrzędna, bazowa). Pomiędzy klasami można określić relację nadrzędności. Jeżeli dwie klasy są ze sobą w relacji, to jedna jest rozszerzeniem drugiej. Oznacza to między innymi, że klasa rozszerzająca ma wszystkie sloty klasy rozszerzanej.

- Obiekty. Nowe instancje klas tworzone są funkcją `methods::new()`. Obiekty tworzone są na bazie prototypu, który musi być określony dla każdej klasy. Zawartość obiektu jest inicjowana funkcją `initialize()` danej klasy.

Przedstawmy kilka przykładów działania na klasach S4. Więcej informacji o klasach S4 można znaleźć w [6]. W tabeli 2.5 przedstawiono wybrane funkcje do pracy na klasach S4.

Rozpocznijmy od definicji nowej klasy, której obiekty będą miały dwa pola: `imie` i `wiek`.

```
setClass("osoba", representation(imie="character",
                                   wiek="numeric"))
## [1] "osoba"
```

W powyższej deklaracji określiliśmy nazwę klasy oraz strukturę, czyli reprezentację klasy. Powyższa przykładowa klasa składa się z dwóch slotów przechowujących odpowiednio łańcuch znaków i liczbę. Możemy teraz tworzyć nowe obiekty tej klasy.

```
(obiektOsoba <- new("osoba", imie="Przemek", wiek=27))
## An object of class “”osoba
## Slot "imie":
## [1] "Przemek"
## Slot "wiek":
## [1] 27
```

Na slotach klasy możemy wykonywać dowolne operacje, póki są one zgodne z deklaracją klasy. Dostęp do slotów jest możliwy dzięki operatorowi `@` oraz metodzie `slot()`.
Poprawna operacja i niepoprawna operacja. Liczba 13 to nie jest łańcuch znaków!.

```
obiektOsoba@imie <- "Przemyslaw"
obiektOsoba@imie <- 13
## Error in checkSlotAssignment(object, name, value) :
##   assignment of an object of class "numeric" is not valid
```

Klasa może zawierać również funkcje. Szczególnie przydatne jest określenie funkcji `initialize()`, która wywoływana jest przy inicjacji nowego obiektu klasy. Poniżej przedstawiamy przykład definicji funkcji, wymuszającej przy inicjacji by wiek przyjmował rozsądne wartości.

Oto jak wygląda nowy konstruktor obiektów klasy `osoba`.

```
setMethod("initialize", "osoba",
     function(.Object, imie = character("Nieznane"), wiek = numeric(0)) {
        if (nargs() > 1) {
            if(wiek > 120 || wiek < 0)
                stop("Niedopuszczalna wartosc wieku")
              .Object@imie <- imie
              .Object@wiek <- wiek
           }
        .Object
     })
```

Sprawdźmy teraz, co się stanie przy inicjacji jeżeli podamy niepoprawną wartość dla zmiennej `wiek`.

```
obiektOsoba <- new("osoba", imie="Przemek", wiek=2700)
## Error in .local(.Object, ...) : Niedopuszczalna ścwarto wieku
##
```

Klasy można łączyć w zbiory klas. Obiektem takiego zbioru może być obiekt dowolnej z klas składowych. Jest to wygodny mechanizm, gdy chcemy, aby np. slot

mógł przyjmować wartości różnych typów. Zbiory klas można tworzyć funkcją `methods::setClassUnion()`. Zdefiniujmy taki przykładowy zbiór przechowujący liczby lub łańcuchy znaków.

```
setClassUnion("liczbaNapis", c("numeric", "character"))
```

Możemy teraz zmodyfikować deklaracje klasy `osoba`, tak by umożliwić podawanie wieku nie tylko liczbowo, ale też opisowo. Poniżej pole `wiek` może być liczbą lub napisem.

```
setClass("osoba2", representation(imie="character", wiek="liczbaNapis"))
obiektOsoba2 <- new("osoba2", imie="Przemek", wiek="dorosly")
```

**Tabela 2.5:** Funkcje z pakietu `methods` do operacji na klasach lub obiektach klas

| | |
|---|---|
| `slotNames(x)` | Argumentem może być nazwa kasy lub jej obiekt. Wynikiem tej funkcji są informacje o nazwach slotów w tej klasie. |
| `isClassUnion(class)` | Test czy klasa `class` jest zbiorem klas. |
| `is(object, class2)` | Test czy obiekt `object` jest obiektem klasy `class2` lub jej podklasy. |
| `as(object, class2)` | Konwersja obiektu `object` na obiekt klasy `class2`. |
| `extends(class1, class2)` | Czy klasa `class1` jest rozszerzeniem `class2`? |

### 2.1.9 Formuły

Formuła to symboliczny opis zależności pomiędzy zmiennymi. Formuły są najczęściej wykorzystywane do opisu postaci modelu liniowego, logistycznego lub innego (patrz rozdział poświęcony statystyce). Formuły mogą być też wykorzystywane do wskazania zbioru zmiennych, które mają być graficznie przedstawione lub które mają być poddane jakiejś operacji. Obiekty zawierające formuły mogą być argumentami wielu funkcji statystycznych i statystyk opisowych. Wiele funkcji graficznych, patrz np. wykres rozrzutu, wykres pudełkowy itp., umożliwia podanie argumentów wejściowych w postaci zbioru wektorów lub w postaci formuły. Podsumowując, formuły to często wykorzystywany sposób opisywania zależności pomiędzy zmiennymi.

Definicja prawie jak stwierdzenie, że dzida składa się z przeddzidiza, śróddzidzia i zadzidzia.

Składnia formuły jest następująca:

```
Lewa.strona ~ Prawa.strona
```

`Lewa.strona` to najczęściej jedna zmienna (nazywana zmienną objaśnianą) lub wyrażenie, w wyrażeniu można wymienić również większą liczbę zmiennych, `Prawa.strona` to jedna lub więcej zmiennych rozdzielanych znakiem `+` (zmienne objaśniające). Zmienne mogą być też rozdzielane innymi znakami, o czym napiszemy później, specjalne znaczenie ma znak `:` oznaczający interakcję pomiędzy zmiennymi. Poniżej przedstawiamy kilka przykładowych formuł.

Trzy przykłady. Formula opisująca zależność pomiędzy dwiema zmiennymi. Formula opisująca addytywną zależność y od dwóch zmiennych. Formula opisująca addytywną zależność od dwóch zmiennych i ich interakcji .

```
y ~ a
y ~ a + b
y ~ a + b + a:b
```

Poniżej przedstawione są wybrane symbole, które można stosować w formułach. W większości przypadków są one rozwijane do kilku wyrazów rozdzielanych symbolem `+`.

- Znak `*` rozdzielający zmienne jest interpretowany jako efekty addytywne wskazanych zmiennych i interakcje pomiędzy zmiennymi, czyli stosowane jest następujące rozwinięcie `a*b = a+b+a:b`.
- Znak `%in%` rozdzielający zmienne jest interpretowany jako zagnieżdżenie zmiennej po lewej stronie w zmiennej po prawej, symbol ten jest rozwijany zgodnie z regułą: `(a+b) %in% c = a:c + b:c`.
- Znak `/` ma podobne znaczenie jak `%in%`, przykładowo: `a/b = a + a:b`.
- Element `^n`, gdzie $n$ to liczba naturalna, jest interpretowany jako lista efektów addytywnych i wszystkie możliwych interakcji do `n`-tego rzędu, przykładowo stosowane jest następujące rozwinięcie

  `(a+b+c)^2 = (a+b+c)*(a+b+c) = a + b + c + a:b + a:c + b:c`.
- Znak `-` przed zmienną powoduje usunięcie wskazanej zmiennej z formuły, przykładowe rozwinięcie formuły: `a*b - a = b + a:b`.

Specjalne znaczenie ma wyraz `"-1"`, który powoduje usunięcie z modelu wyrazu wolnego. Identyczny efekt można otrzymać dodając wyraz `"+0"`. Jeżeli w formule chcemy jawnie wskazać lub dodać wyraz wolny to można to zrobić dodając wyraz `+1`.

W formule można wykorzystywać funkcje arytmetyczne i inne funkcje programu R. Poprawnymi formułami są:

```
log(y) ~ a + log(b)
y ~ exp(a + b)
y ~ sqrt(b) + a
y ~ ifelse(a>0, b, c)
```

Można więc w formułach używać wyrażeń arytmetycznych, problem może się pojawić, jeżeli chcemy skorzystać z operatorów arytmetycznych, które w formułach mają specjalne znaczenie. Znak `+` w formule zostanie zinterpretowany jako dodanie wyrazu do formuły, a nie jak operację dodawania. Aby temu zapobiec, można skorzystać z funkcji `I()`, oznaczającej identyczność. Argumenty tej funkcji traktowane będą jako wyrażenia arytmetyczne a nie elementy formuły.

W pierszym przykładzie formuła opisuje zależność y od zmiennych a i b, w kolejnym formuła opisująca zależność y od zmiennej suma a i b. Ostatni przykład to bardziej skomplikowana formuła o dwóch wyrazach.

```
y ~ a+b
y ~ I(a+b)
y ~ I(a+b) + I(a*(b-3))
```

Bardzo przydatnym wyrazem jest "." (kropka), który oznacza wszystkie pozostałe kolumny ramki danych (zakładając, że w funkcji w której podaliśmy formułę występuje argument `data` wskazujący ramkę danych). Przykłady formuł z tym wyrazem umieszczone są poniżej. Są to: Zależność y od wszystkich pozostałych zmiennych; Zależność y od wszystkich pozostałych zmiennych, bez wyrazu wolnego; Zależność y od wszystkich zmiennych plus nowa zmienna iloraz a i b.

```
y ~ .
y ~ . -1
y ~ . + I(a/b)
```

Nazwy zmiennych pojawiających się w formule powinny być widoczne w aktualnym środowisku lub też powinny wskazywać kolumny ramki danych, wskazanej przez dodatkowy argument. Poniższe trzy przykłady dają identyczny wynik, niezależnie od tego czy: podajemy do formuły nazwy zmiennych; podajemy do formuły nazwy kolumn we wskazanej ramce danych; każemy wybrać wszystkie kolumny ze wskazanej ramki danych.

```
y <- rnorm(100)
x <- rnorm(100)
dane <- data.frame(a=y, b=x)

lm(y~x)
lm(a~b, data=dane)
lm(a~., data=dane)
```

Przykłady wykorzystania formuł znajdują się w rozdziale 3.4 oraz w podrozdziałach poświęconych grafice.

## 2.2 Przetwarzanie potokowe

Analizując dane bardzo często spotykamy się z sytuacją, gdy chcemy wyznaczyć wartość pewnej funkcji dla wszystkich kolumn macierzy lub ramki danych, albo dla wszystkich wierszy, albo dla wszystkich grup wierszy określonych przez pewną zmienną grupującą albo różnych zestawów parametrów.

Jeżeli potrafimy posługiwać się pętlami, to z pewnością w takich sytuacjach będziemy potrafili napisać odpowiednią pętlę lub kilka pętli, które policzą to co policzyć chcemy. Ale ponieważ tego typu operacje są bardzo częste w analizie danych, a program R powstał z myślą o analizie danych, dlatego też są w nim dostępne unikalne rozwiązania, pozwalające na łatwe wyznaczanie wartości funkcji w podgrupach, podwymiarach zbioru danych.

Jest wiele mechanizmów wspierających przetwarzanie w modelu zastosuj zadaną funkcję na zbiorze różnych danych wejściowych. Poniżej przedstawimy dwa najpopularniejsze. Pierwszy, to skorzystanie z funkcji z bogatej klasy funkcji `*apply`. Drugi, to przetwarzanie z użyciem pakietów `plyr` i `reshape2`.

Pakiet reshape2 to nowa wersja pakietu reshape. Autor tych pakietów, Hadley Wickham, uznał że należy zmienić sposób działania, ale nie chciał też zmuszać użytkowników starej wersji do zmiany swoich programów, dlatego zdecydował się na udostępnienie nowej wersji pod zmienioną nazwą. Przykładowo funkcja cast() nie jest już publicznie dostępna, zamiast niej korzysta się z acast() lub dcast().

### 2.2.1 Pakiety plyr i reshape2

W podejściu z użyciem pary plyr+reshape2, pakiet reshape2 służy to zmiany struktury zbioru danych, a pakiet plyr służy do automatyzacji obliczeń, na odpowiednio przygotowanej strukturze danych. Poniżej przedstawiam krótkie wprowadzenie. Osobom zainteresowanym poznaniem dalszych szczegółów polecam prace [46] i [47].

Zacznijmy od przedstawienia pakietu reshape2. W tym pakiecie godne uwagi są trzy funkcje: reshape2::melt(), reshape2::acast() i reshape2::dcast(). Funkcja melt() „roztapia” dane z postaci tabelarycznej (również dla tabel wielowymiarowych) do tzw. postaci wąskiej (nazywanej też postacią rzadką lub postacią wiersz-kolumna-wartość (RCV od ang. *row column value*). Funkcja acast() pozwala na przekształcenie danych z postaci wąskiej na postać tabeli krzyżowej (przestawnej), gdzie wartościami pól tej tabeli są wartości zadanej funkcji wyznaczonej na podzbiorze danych.

Przedstawmy obie funkcje na przykładzie. Wykorzystamy pakiet SmarterPoland do pobrania z internetowych baz Eurostatu danych o częstości wypadków drogowych w różnych latach, różnych krajach. Takie dane znajdują się w tabeli tsdtr420 w bazie Eurostatu.

Poniżej wykorzystujemy funkcję SmarterPoland::getEurostatRCV() do pobrania danych z tabeli tsdtr420.

```
library(SmarterPoland)
wypadkiWaska <- getEurostatRCV("tsdtr420")
## trying URL 'http://epp.eurostat.ec.europa.eu/NavTree_prod/everybody/
    BulkDownloadListing?sort=1&file=data%2Ftsdtr420.tsv.gz'
## Content type 'application/x-gzip' length 2355 bytes
## opened URL
## ==================================================
## downloaded 2355 bytes
```

Kilka pierwszych wierszy z pobranych danych. W kolejnych kolumnach są: nazwa opisywanego współczynnika, identyfikator kraju, roku oraz liczba ofiar wypadków.

```
head(wypadkiWaska, 4)
##   victim geo time value
## 1    KIL  AT 1991  1551
## 2    KIL  BE 1991  1873
## 3    KIL  BG 1991  1114
## 4    KIL  CY 1991   103
```

Podsumowanie danych z częstościami wystąpień zmiennych czynnikowych.

```
summary(wypadkiWaska)
##          victim          geo          time             value
##  KIL        :532   AT     : 38   1991   : 56   Min.   :    4.0
##  KIL_MIO_POP:532   BE     : 38   1992   : 56   1st Qu.:  117.0
##                    BG     : 38   1993   : 56   Median :  204.5
##                    CY     : 38   1994   : 56   Mean   : 2070.9
##                    CZ     : 38   1995   : 56   3rd Qu.: 1066.2
##                    DE     : 38   1996   : 56   Max.   :75426.0
##                    (Other):836   (Other):728   NA's   :28
```

Zmienna `wypadkiWaska` jest już w postaci roztopionej. Pierwsze trzy kolumny opisują trzy różne wymiary. Tutaj są to: rodzaj statystyki (liczba ofiar i liczba ofiar przeliczona na milion mieszkańców), identyfikator kraju, rok, którego dotyczy statystyka. Czwarta kolumna określa wartość zadanej statystyki dla wybranego kraju w wybranym roku. W tym przypadku postać wąska opisuje trzy wymiary zmiennych, ale można rozważać zarówno dane o większej jak i o mniejszej liczbie wymiarów.

Funkcja `dcast()` od `acast()` różni się wyłącznie formatem wyjścia. W przypadku pierwszej wyjście to ramka danych a w przypadku drugiej macierz (ramka danych nie może mieć więcej niż dwóch wymiarów, macierz może).

Wykorzystajmy teraz funkcję `dcast()` aby przekształcić te dane do postaci tabelarycznej z krajami w wierszach i latami w kolumnach. Przyjmijmy, że interesuje nas wyłącznie statystyka `KIL_MIO_POP` („ofiary na milion mieszkańców").

```
wypadkiSzeroka <- dcast(wypadkiWaska, geo ~ time, mean, na.rm=TRUE,
                        subset = .(victim == "KIL_MIO_POP"))
```

Pierwszym argumentem jest ramka danych w postaci roztopionej. Drugim jest formuła opisująca, które wymiary chcemy by były w wierszach (lewa strona formuły) a które w kolumnach (prawa strona formuły). Argument `subset` określa, które wiersze z wejściowego zbioru danych brać pod uwagę. Trzeci argument wskazuje funkcję, która będzie zastosowana do wszystkich wierszy ramki `wypadkiWaska` wskazujących na ten sam rok i kraj. W rozważanym przypadku rok i kraj wyznaczają dokładnie jeden wiersz, więc użycie funkcji `mean()` może wydawać się nadmiarowe, za każdym razem jest to bowiem liczenie średniej z jednej wartości.

```
dim(wypadkiSzeroka)
## [1] 28 20
wypadkiSzeroka[wypadkiSzeroka$geo %in%
        c("UK", "SK", "FR", "PL", "ES", "PT", "LV"),1:10]
##    geo 1991 1992 1993 1994 1995 1996 1997 1998 1999
## 10  ES  227  200  163  143  146  139  142  150  144
## 13  FR  184  173  172  157  154  147  145  153  145
## 19  LV  375  298  280  305  264  241  232  280  272
## 22  PL  207  181  165  175  179  165  189  183  174
## 23  PT  323  310  271  251  271  272  250  210  200
## 27  SK  116  128  110  119  123  115  146  152  120
## 28  UK   83   76   69   66   65   64   64   61   61
```

Dla ćwiczeń roztopimy teraz zbiór danych `wypadkiSzeroka`, sprowadzając go z powrotem do postaci wąskiej z użyciem funkcji `melt()`.

```
wypadkiRoztopiona <- melt(wypadkiSzeroka, id = "geo", measure=paste
    (1991:2008))
head(wypadkiRoztopiona)
##          geo value time
## X1991     AT   201 1991
## X1991.1   BE   188 1991
## X1991.2   BG   129 1991
## X1991.3   CY   175 1991
## X1991.4   CZ   129 1991
## X1991.5   DE   142 1991
```

Teraz omówmy wybrane, przydatne funkcje z pakietu `plyr`. Tabela 2.2.1 przedstawia nazwy wszystkich interesujących funkcji. Jednak nie będziemy ich wszyst-

| wejście \ wyjście | macierz | ramka danych | lista | nic |
|---|---|---|---|---|
| **macierz** | `aaply()` | `adply()` | `alply()` | `a_ply()` |
| **ramka danych** | `daply()` | `ddply()` | `dlply()` | `d_ply()` |
| **lista** | `laply()` | `ldply()` | `llply()` | `l_ply()` |
| **n powtórzeń** | `raply()` | `rdply()` | `rlply()` | `r_ply()` |
| **zbiór argumentów** | `maply()` | `mdply()` | `mlply()` | `m_ply()` |

**Tabela 2.6:** Wybrane funkcje z pakietu `plyr` do przetwarzania potokowego. W kolumnach zaznaczono jakiego typu jest wynik funkcji, czy jest to macierz, ramka danych, lista czy też funkcja nie zwraca wartości. W wierszach zaznaczono co jest argumentem wejściowym, czy jest to macierz danych, ramka danych, lista zbiorów danych.

kich omawiać, ponieważ ich sposób działania jest bardzo podobny i wystarczy omówić kilka wybranych przykładów by wiedzieć jak korzystać z pozostałych.

Schemat działania funkcji z tego pakietu jest określany „dziel/przekształć/łącz" (ang. *split/apply/combine*). Przetwarzanie potokowe oznacza, że ta sama funkcja jest stosowana do różnych podzbiorów wejściowego zbioru danych. W pierwszym kroku ten zbiór danych jest dzielony na podzbioru najczęściej z uwagi na wybrane zmienne grupujące (etap ang. *split*), następnie do każdego wydzielonego podzbioru stosowana jest zadana funkcja (etap ang. *apply*) w ostatnim kroku otrzymane wyniki są łączone (etap ang. *combine*).

Konwencja nazywania funkcji jest ujednolicona, pierwsza litera odpowiada formatowi argumentów wejściowych, druga formatowi wyniku następnie występuje sufiks `ply()`. Litera `a` oznacza macierz (*array*), `d` ramkę danych, `l` listę, `_` nic (funkcji, które nie zwracają wyniku używa się dla ich „efektów ubocznych", np rysowania wykresów). Prefiks `r` oznacza powtórzenie pewnej operacji `n`-krotnie na całym zbierze danych, a prefiks `m` oznacza powtórzenie tej samej funkcji dla zadanego zbioru różnych argumentów wejściowych.

W przypadku pierwszy trzech wierszy tabeli funkcje prezentowane różnią się typem wejścia i wyjścia. Mechanizm działania jest podobny. Poniżej przedstawimy tylko funkcje `ddply()` i `llply()`. Argumenty tych funkcji są następujące: zbiór danych wejściowych, wskazanie zmiennych grupujących, wskazanie funkcji do zastosowania na każdym podzbiorze, dodatkowe argumenty dla tej funkcji. Zmienne grupujące można wskazać na różne sposoby: podając wektor nazw, formułę lub używając funkcji `.()`, która nic nie robi ale umożliwia podanie jako argumentów nazw kolumn ze zbioru danych.

W przykładzie poniżej zbiór danych `wypadkiRoztopiona` podzielimy ze względu na podzbiory określone zmienną `geo` (czyli podzbiory danych to dane dla różnych krajów). Następnie na każdym podzbiorze wykonamy funkcję `lm()`, która dla każdego podzbioru wyznaczy współczynniki regresji liniowej - zależności liczby wypadków od roku. Zarówno wejściem jak i wyjściem funkcji jest ramka danych.

```
wynik <- ddply(wypadkiRoztopiona, .(geo), function(x)
                    lm(value ~ as.numeric(time), data=x)$coef)
head(wynik)
##   geo (Intercept) as.numeric(time)
## 1  AT    186.6993        -6.097007
```

```
## 2  BE    180.1503        -4.793602
## 3  BG    142.1307        -1.054696
## 4  CY    212.8039        -5.646027
## 5  CZ    158.0719        -2.223942
## 6  DE    139.9216        -4.868937
```

Często używając funkcji z rodziny `**ply()` wykorzystuje się funkcje `summarize()` i `transform()`. Obie pozwalają na uniknięcie potrzeby tworzenia funkcji anonimowych. Funkcja `summarize()` jako wynik zwraca wskazane podsumowania każdego podzbioru danych (w poniższym przypadku będą to współczynniki modelu liniowego i średnia liczba wypadków). Funkcja `transform()` jako wynik zwraca przekształcony zbiór danych z dodanymi nowymi kolumnami.

```
wynik <- ddply(wypadkiRoztopiona, .(geo), summarize,
        zmiana = lm(value ~ as.numeric(time))$coef[2],
        pprzeciecia = lm(value ~ as.numeric(time))$coef[1],
        srednia = mean(value, na.rm=T))
head(wynik)
##   geo    zmiana pprzeciecia   srednia
## 1  AT -6.097007    186.6993 128.77778
## 2  BE -4.793602    180.1503 134.61111
## 3  BG -1.054696    142.1307 132.11111
## 4  CY -5.646027    212.8039 159.16667
## 5  CZ -2.223942    158.0719 136.94444
## 6  DE -4.868937    139.9216  93.66667
```

Poniżej zaprezentujemy przykład użycia funkcji `llply()`. Oczekuje ona jako argumentu wejściowego listy. Aby taką listę otrzymać posłużymy się funkcją `split()`.

```
lista <- split(wypadkiRoztopiona, wypadkiRoztopiona$geo)
head(lista[[1]])
##       geo value time
## X1991  AT   201 1991
## X1992  AT   180 1992
## X1993  AT   163 1993
## X1994  AT   169 1994
## X1995  AT   152 1995
## X1996  AT   129 1996
```

Użycie funkcji `llply()` jest identyczne z funkcją `ddply()`, z tą różnicą, że wejście i wyjście jest teraz listą.

```
wynik <- llply(lista, function(x)
             lm(value ~ as.numeric(time), data=x)$coef)
wynik[[1]]
##      (Intercept) as.numeric(time)
##       186.699346        -6.097007
```

Ciekawą funkcją, którą omówimy poniżej jest `r*ply()`. Powtarza ona wykonanie pewnej operacji zadaną liczbę razy, przez co działa podobnie co funkcja `replicate()` ale mamy większą kontrolę nad typem wyniku.

```
wierszy <- nrow(wypadkiRoztopiona)
wynik <- rdply(100, lm(value ~ as.numeric(time), data=wypadkiRoztopiona[
    sample(wierszy, replace=TRUE),])$coef)
```

```
head(wynik, 3)
##   .n (Intercept) as.numeric(time)
## 1  1    169.8856        -4.315885
## 2  2    178.8349        -4.663816
## 3  3    171.0432        -4.247858
summary(wynik)
##        .n          (Intercept)    as.numeric(time)
##  Min.   :  1.00   Min.   :162.5   Min.   :-5.816
##  1st Qu.: 25.75   1st Qu.:168.6   1st Qu.:-4.736
##  Median : 50.50   Median :172.5   Median :-4.437
##  Mean   : 50.50   Mean   :173.0   Mean   :-4.465
##  3rd Qu.: 75.25   3rd Qu.:176.2   3rd Qu.:-4.063
##  Max.   :100.00   Max.   :187.9   Max.   :-3.567
```

### 2.2.2 Rodzina funkcji *apply

Rodzina funkcji `*apply` ma bardzo zbliżone możliwości do funkcji `**ply()`. Większość funkcji z tej rodziny wymienionych jest w tabeli 2.7. Wszystkie one pozwalają na wykonanie pewnej operacji na szeregu podzbiorów danych. Operacja, która ma być wykonana określana jest przez argument `FUN`. Różnica pomiędzy funkcjami z tej rodziny polega na sposobie definiowania podzbiorów danych, na których ta funkcja ma być wykonana. Te podzbiory mogą być elementami listy (patrz `lapply()`), elementami wektora (patrz `sapply()`), macierzy (patrz `apply()`), podgrup wskazanych przez jedną lub kilka zmiennych (patrz `by()` i `tapply()`).

Zacznijmy od przedstawienia funkcji `lapply()`. Spodziewa się ona przynajmniej dwóch argumentów `X` i `FUN`. Jej użycie pozwala na wykonanie funkcji `FUN`, dla każdego elementu listy `X` (często wygodnie jest w miejsce argumentu `FUN` użyć funkcji anonimowych, przedstawimy odpowiedni przykład poniżej). Wynikiem działania funkcji `lapply()` jest lista, której poszczególne elementy są wynikami działania funkcji `FUN` na kolejnych elementach listy `X`. Jeżeli zostaną podane dodatkowe argumenty do funkcji `lapply()`, to zostaną one przekazane do funkcji `FUN`. Dzięki czemu możemy korzystać z funkcji `FUN` przyjmujących wiele argumentów.

Użyteczną funkcją z tej rodziny jest też `rollapply()` z pakietu zoo. Pozwala ona na wyznaczenie wartości określonej funkcji na oknie o określonej szerokości przesuwającym się wzdłuż wektora, lub szeregu czasowego. Bardzo wygodna do wygładzania, uśredniania „w oknie”, czy śledzenia jak zmienia się lokalny trend.

Domyślnie wynikiem funkcji `lapply()` jest lista, ale jeżeli w wyniku chcielibyśmy otrzymać wektor, to jednym z rozwiązań jest zamiana listy na wektor funkcją `unlist()`. Jeżeli jako wynik chcemy otrzymać wektor lub macierz, zamiast listy, możemy użyć funkcji `sapply()`, która działa podobnie do `lapply()`, ale ma argument `simplify`, który jeżeli ma wartość `TRUE` (domyślnie) i wszystkie elementy listy są tego samego typu, to wynik jest upraszczany do wektora lub macierzy.

Więcej funkcji z rodziny `*apply()` wymienionych jest w tabeli 2.7. Poniżej przedstawiamy przykłady użycia tych funkcji. Zauważmy, że tę samą operację można zazwyczaj wykonać używając funkcji `*apply` lub pętli `for`. Z reguły czas działania funkcji z rodziny `*apply` jest krótszy a i zapis czytelniejszy. W przykładach poniżej ponownie wykorzystamy zbiór danych `wypadkiWaska`, zdefiniowany w poprzednim podrozdziale.

```
head(wypadkiWaska, 3)
##   victim geo time value
## 1    KIL  AT 1991  1551
```

```
## 2    KIL  BE 1991  1873
## 3    KIL  BG 1991  1114
```

Funkcją `tapply()` liczymy średnią liczbę wypadków w grupach określonych przez zmienną geo. Dodatkowy argument `na.rm=TRUE` zostanie przekazany do funkcji `mean()`.

```
tapply(wypadkiWaska$value, wypadkiWaska$geo, mean, na.rm=TRUE)
##          AT          BE          BG          CY          CZ
##   580.45946   760.72973   610.37838   131.89189   776.13514
##          DK          EE          EL          ES        EU27
##   282.21622   208.05405  1071.97297  2824.13514 28192.43243
## ...
```

A teraz policzymy średnie funkcją `by()`. Po usunięciu atrybutu `class` wynik tej funkcji to macierz liczb. Funkcja `by()` pozwala (w przeciwieństwie do `tapply()`) aby pierwszy i drugi argument wskazywały więcej niż jedną zmienną.

```
by(wypadkiWaska$value, wypadkiWaska$geo, mean, na.rm=TRUE)
## wypadkiWaska$geo: AT
## [1] 580.4595
## --------------------------------------------------------
## wypadkiWaska$geo: BE
## [1] 760.7297
## --------------------------------------------------------
## wypadkiWaska$geo: BG
## [1] 610.3784
```

I funkcją `aggregate()` otrzymujemy te same wyniki ale w innej formie.

```
aggregate(wypadkiWaska$value,
          list(wypadkiWaska$geo, wypadkiWaska$victim), mean, na.rm=TRUE)
##    Group.1     Group.2          x
## 1       AT         KIL  1008.36842
## 2       BE         KIL  1353.89474
## 3       BG         KIL  1063.47368
```

Funkcja `eapply()` pozwala iterować po obiektach w przestrzeni nazw. Wynikiem poniższej instrukcji jest wektor z rozmiarami obiektów z przestrzeni nazw.

```
unlist(eapply(environment(), FUN=object.size))
##        grupy podpopulacje        cecha
##          424          424          432
```

Funkcją `sapply()` wykonuje operację na każdym elemencie wektora, w tym przykładzie sprawdzamy czy liczba jest w przedziale >`1000`.

```
sapply(wypadkiWaska$value, FUN = function(x) (x  > 1000))
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [9]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [17]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE FALSE
...
```

Funkcja `apply()` wykonuje określoną operacją dla każdego wektora lub kolumny wejściowej ramki danych. Poniżej wyznaczamy elementy minimalny i maksymalny w kolumnach.

```
apply(wypadkiWaska, 2, range, na.rm=TRUE)
##      victim        geo  time    value
## [1,] "KIL"         "AT" "1991"  "    4"
## [2,] "KIL_MIO_POP" "UK" "2009 " "75426"
```

**Tabela 2.7:** Funkcje z rodziny `*apply()` z pakietu `base`

| | |
|---|---|
| `tapply(x,index,fun)` | Wykonuje funkcję `fun` dla podzbiorów wektora `x` określonego przez poziomy zmiennej czynnikowej `index`. Przydatna funkcja, gdy chcemy policzyć pewną statystykę w podgrupach, np. średni wiek w podziale na płeć. W tym przypadku `x` będzie wektorem z wiekiem, `index` wektorem z płcią a `fun` będzie funkcją `mean`). |
| `sapply(x,fun,...)`, `lapply(x,fun,...)` | Wykonuje funkcję `fun` dla wszystkich elementów wektora `x`. Przydatna funkcja zastępująca pętlę `for`. |
| `by(x, index, fun)` | Bardziej potężna wersja funkcji `tapply()` z tą różnicą, że `x` może być macierzą lub listą, index może być listą, a wynik tej funkcji jest specyficznie wyświetlany. Jeżeli `index` jest listą zmiennych czynnikowych, to wartość funkcji `fun` będzie wyznaczona dla każdego przecięcia czynników tych zmiennych. Wynik funkcji `by()` jest klasy `by`, ale po usunięciu informacji o klasie, np. poprzez użycie funkcji `unclass()` otrzymujemy zwykłą macierz. Argument `x` może być listą lub macierzą, dzięki czemu do funkcji `fun` przekazać można kilka zmiennych – elementów/kolumn listy/macierzy `x`. |
| `mapply(fun, ...)` | Wielowymiarowy odpowiednik funkcji `sapply()`. Argumentami tej funkcji jest funkcja `fun` oraz kilka (dwa lub więcej) wektorów o tej samej długości. Wynikiem jest wektor, w którego pozycji `i` jest wynik funkcji `fun` wywołanej z argumentami określonymi przez elementy o indeksach `i` pozostałych argumentów funkcji `mapply()`. |
| `rapply(x, fun)` | Rekursywna wersja `lapply()` przydatna, gdy mamy do czynienia z listą list lub innymi, zagnieżdżonymi strukturami. |
| `eapply(env, fun)` | Aplikuje funkcję `fun` do każdego elementu wskazanego środowiska (przestrzeni nazw) `env`. |
| `aggregate(x, by, fun)` | Wykonuje funkcję `fun` dla kolumn macierzy `x` określonego przez poziomy zmiennych czynnikowych zapisanych w liście `by`. Podobna w działaniu do `tapply()`, przy czym argument `x` może być macierzą (w `tapply()` był wektorem) a grupowanie w podpopulacjach może być określone przez więcej niż jedną zmienną (argument `by` może określać więcej zmiennych grupujących). |
| `replicate(n,expr)` | Ta funkcja powoduje `n`-krotne wykonanie polecenia `expr`. Wynikiem jest lista, której kolejne elementy zawierają wartość zwróconą w wyniku wywołania wyrażenia `expr`. Przydatna w sytuacji gdy w `expr` jest randomizacja (np. generowanie rozkładu statystyki testowej). |

## 2.3 Wybrane funkcje matematyczne

W programie R znaleźć można wiele specjalizowanych, mniej i bardziej popularnych funkcji matematycznych. W poniższych podrozdziałach przedstawimy kilka przykładowych rodzin funkcji. Uczciwie zaznaczamy, że jest to jedynie wycinek znacznie większego zbioru funkcji matematycznych dostępnych w programie R.

### 2.3.1 Wielomiany

W pakiecie `polynom` zebrane są funkcje do tworzenia i operowania na wielomianach. W tabeli 2.8 przedstawione są wybrane funkcje z tego pakietu a poniżej przedstawiono przykład wywołania kilku z nich.

Zacznijmy od zdefiniowania dwóch przykładowych wielomianów.

```
(p1 <- polynomial(c(2,0,1)))
## 2 + x^2
(p2 <- polynomial(c(2,2,1,1)))
## 2 + 2*x + x^2 + x^3
```

Wielomiany możemy dodawać, odejmować, mnożyć, dzielić używając standardowych operatorów.

```
p1 + p2
## 4 + 2*x + 2*x^2 + x^3
p1 * p2
## 4 + 4*x + 4*x^2 + 4*x^3 + x^4 + x^5
```

Wielomiany możemy całkować i różniczkować

```
integral(p1,c(0,1))
## [1] 2.333333
deriv(p2)
## 2 + 2*x + 3*x^2
```

Definiować je wskazując ich zera...

```
poly.calc(c(-1,1))
## -1 + x^2
```

...lub wskazując punkty, które ten wielomian ma zawierać.

```
poly.calc(c(0,2,4), c(3,2,3))
## 3 - x + 0.25*x^2
```

Możemy wyznaczać zera wielomianu.

```
solve(p2)
## [1] -1+0.000000i  0-1.414214i  0+1.414214i
```

Najmniejsza wspólna wielokrotność i największy wspólny dzielnik.

```
LCM(p1,p2)
## 2 + 2*x + x^2 + x^3
GCD(p1,p2)
## 2 + x^2
```

### 2.3.2 Bazy wielomianów ortogonalnych

W pakiecie `orthopolynom` zebrane są funkcje tworzące wielomiany z bazy wielomianów ortogonalnych. W pakiecie tym dostępnych jest wiele baz wielomianów, np. Czebyszewa, Gegenbauera, Hermite'a, Laguerre'a, Jacobiego i wiele innych. Poniżej przedstawiamy przykład kilku operacji na wielomianach Legendre'a. Funkcja `orthopolynom::legendre.polynomials()` wyznacza elementy bazy wielomianów na odcinku `[-1,1]`, podczas gdy funkcja `slegendre.polynomials()` wyznacza elementy bazy wielomianów na odcinku `[0,1]`. Pierwszym argumentem tych funkcji jest maksymalny stopień wielomianu, drugim argumentem (o nazwie `normalized`) jest informacja, czy wielomiany mają być normalizowane czy nie, czyli czy chcemy mieć bazę ortogonalną czy ortonormalną. Poniżej przedstawiamy przykładowe operacje na wielomianach Legendre'a i wykres dla pierwszych pięciu wielomianów.

Wyznaczamy pierwsze pięć wielomianów Legendre'a. Za bazę wybieramy odcinek [0,1], a wielomiany chcemy mieć normalizowane.

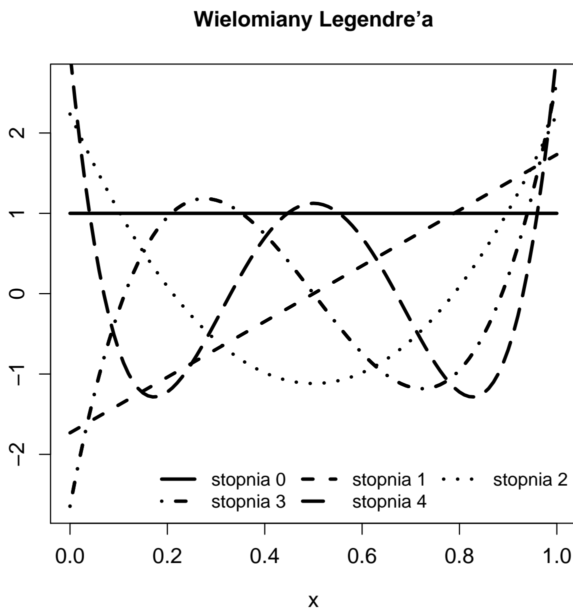
```
library(orthopolynom)
Nmax <- 5
(wielomiany <- slegendre.polynomials(Nmax, normalized=TRUE))
## [[1]]
## 1
## [[2]]
## -1.732051 + 3.464102*x
## [[3]]
## 2.236068 - 13.41641*x + 13.41641*x^2
## [[4]]
## -2.645751 + 31.74902*x - 79.37254*x^2 + 52.91503*x^3
## [[5]]
## 3 - 60*x + 270*x^2 - 420*x^3 + 210*x^4
## [[6]]
## -3.316 + 99.498*x - 696.491*x^2 + 1857.31*x^3 - 2089.474*x^4 + 835.789
    *x^5
```

Konwertujemy wielomian na funkcję programu R. Z obiektu `wielomian4` można teraz korzystać jak ze zwykłej funkcji w programie R. Wyznaczmy jej wartość w kilku punktach.

```
wielomian4  <- as.function(wielomiany[[4]])
wielomian4(c(0, 0.5, 1))
## [1] -2.645751 0  2.645751
```

Taką funkcję możemy już narysować. Narysujmy też pozostałe wielomiany z rysunku 2.2.

```
curve(wielomian4, 0, 1, lwd=3, lty=4, ylab="")
for (i in 1:Nmax) {
     wielomian <- as.function(wielomiany[[i]])
     curve(wielomian, 0, 1, add=TRUE, lwd=3, lty=i)
}
```




**Rysunek 2.2:** Pierwsze pięć ortogonalnych wielomianów Legendre'a

**Tabela 2.8:** Lista wybranych funkcji z pakietu `polynom`.

| | |
|---|---|
| `polynomial(wsp)` | Ta funkcja pozwala zbudować wielomian przez podanie współczynników tego wielomianu. Współczynniki określa wektor `wsp`, pierwszy element tego wektora to wyraz wolny, kolejny wraz przy elemencie liniowym itp. |
| `integral(pol,lim)` | Ta funkcja wyznacza całkę z wielomianu `pol` w granicach `lim`. |
| `deriv(pol)` | Ta funkcja wyznacza pochodną wielomianu `pol`. |
| `poly.calc(x)`, `poly.calc(x,y)` | Ta funkcja wyznacza wielomian o możliwie najmniejszym stopniu, o zerach w punktach `x` (jeżeli podany będzie tylko jeden argument) lub wielomian przechodzący przez punkty `x,y` (jeżeli podane będą dwa argumenty). |
| `GCD(pol1, pol2)` | Ta funkcja wyznacza największy wspólny dzielnik dwóch wielomianów. |
| `LCM(pol1, pol2)` | A ta najmniejszą wspólną wielokrotność. |
| `solve(pol)`, `polyroot(pol)` | Ta funkcja wyznacza zera danego wielomianu (działa również dla wielomianów zespolonych). |

**Tabela 2.9:** Lista funkcji specjalnych Bessela

| | |
|---|---|
| `besselI(x, nu)` | Zmodyfikowana funkcja Bessela pierwszego rodzaju. |
| `besselK(x, nu)` | Zmodyfikowana funkcja Bessela trzeciego rodzaju. |
| `besselJ(x, nu)` | Funkcja Bessela pierwszego rodzaju. |
| `besselY(x, nu)` | Funkcja Bessela drugiego rodzaju. |

### 2.3.3 Operacje na zbiorach

W poprzednich podrozdziałach przedstawiliśmy przykłady operacji na wektorach, listach i macierzach. W programie R można wykonywać również operacje na zbiorach. Do operacji na zbiorach służą funkcje wymienione w tabeli 2.10. Zbiory są reprezentowane jako wektory.

Poniżej przedstawimy kilka przykładów operacji na zbiorach. Przygotujmy dwa zbiory liczb. Wykonujemy różnicę symetryczną (operację xor), czyli sumę zbiorów minus ich część wspólna.

```
x <- 1:10
y <- 5:15
setdiff(union(x,y), intersect(x,y))
## [1]  1  2  3  4 11 12 13 14 15
```

Sprawdzamy, czy dwa zbiory są równe.

```
setequal(x, setdiff(x, setdiff(y,x)))
## [1] TRUE
```

Sprawdzamy, czy element należy do zbioru.

```
is.element(3, x)
## [1] TRUE
```

**Tabela 2.10:** Funkcje z pakietu `base` do operowania na zbiorach

| | |
|---|---|
| `union(x, y)` | Wynikiem tej funkcji jest suma zbiorów `x` i `y`. |
| `intersect(x, y)` | Wynikiem tej funkcji jest część wspólna zbiorów `x` i `y`. |
| `setdiff(x, y)` | Wynikiem tej funkcji jest różnica zbiór `x` minus `y`. Różnica zbiorów jest funkcją niesymetryczną. |
| `setequal(x, y)` | Wynikiem tej funkcji jest wartość logiczna, równa `TRUE`, gdy zawartości obu zbiorów są sobie równe. |
| `is.element(el, set)` | Wynikiem tej funkcji jest wartość logiczna, równa `TRUE`, gdy element `el` należy do zbioru `set`. |

### 2.3.4 Szukanie maksimum/minimum/zer funkcji

W wielu zagadnieniach statystycznych i nie tylko rozwiązywany problem sprowadza się do szukania zer funkcji lub punktów minimalizujących/maksymalizujących funkcje. W programie R znaleźć można wiele funkcji, które można wykorzystać do tego celu. Poniżej omówimy funkcje `optimize()` (również występująca pod nazwą `optimise()`) i `uniroot()`. Pierwsza z nich wyznacza ekstrema, druga wyznacza miejsca zerowe. W obu przypadkach, jeżeli rozwiązań jest więcej niż jedno, to wyznaczane jest dowolne. Deklaracje tych funkcji są następujące:

```
optimize(f, interval,  ..., lower = min(interval),
       upper = max(interval), maximum = FALSE,
       tol = .Machine$double.eps^0.25)
```

```
uniroot(f,interval,...,lower=min(interval),upper=max(interval),
         f.lower = f(lower, ...), f.upper = f(upper, ...),
         tol = .Machine$double.eps^0.25, maxiter = 1000)
```

Argumentami obu funkcji są: `f` czyli funkcja, dla której szukane mają być ekstrema/miejsca zerowe, `interval` określający na jakim przedziale wyznaczane ma być ekstremum/miejsce zerowe. Dodatkowo, w funkcji `optimize()` można określić, czy poszukujemy maksimum czy minimum (argument `maximum`), a w funkcji `uniroot()` można wskazać maksymalną liczbę iteracji do wykonania (argument `maxiter`). Do badanej funkcji `f` można przekazać również dodatkowe argumenty.

Poniżej przykłady użycia. Zdefiniujemy funkcję do badania `f()`. Szukamy jej minimum na przedziale $[-1, 10]$, określamy też dodatkowy argument dla funkcji `f`, czyli `wyznacznik`.

```
f <- function(x, wyznacznik) {(x - 7)^wyznacznik - x}
optimize(f, interval=c(-1,10), wyznacznik = 2)
## $minimum
## [1] 7.5
## $objective
## [1] -7.25
```

Jak powyżej, ale z użyciem funkcji `uniroot()`.

```
uniroot(f, interval=c(-1,10), wyznacznik = 2)
## $root
## [1] 4.807418
## $f.root
## [1] -4.69523e-06
## $iter
## [1] 8
## $estim.prec
## [1] 6.103516e-05
```

Do maksymalizacji funkcji można też wykorzystać funkcje `optim()` i `nlm()`. W funkcji `optim()` zaimplementowanych jest wiele algorytmów optymalizacji, np. metoda Neldera Meada czy metody wykorzystujące quasi hessiany, więcej informacji znaleźć można przy opisie argumentu `method`. Funkcja `optim()` może być wykorzystywana do optymalizacji po wielowymiarowym zbiorze parametrów, podczas gdy funkcja `optimize()` dotyczyła jednowymiarowego zbioru parametrów.

Więcej o optymalizacji liniowej i nieliniowej można przeczytać w dostępnych w Internecie materiałach, np. [52] i [53].

### 2.3.5 Rachunek różniczkowo–całkowy

Rozdział ten zakończymy przykładami funkcji do różniczkowania i całkowania. Operacje symbolicznego różniczkowania są zaimplementowane w funkcjach `D()` i `deriv()` z pakietu `stats`. Obie funkcje liczą to samo, a różnią się sposobem podania argumentu do zróżniczkowania. W przypadku funkcji `D()` argument powinien być klasy `expression`, a w przypadku funkcji `deriv()` argument powinien być formułą bez określonej lewej strony. Wielomiany można też różniczkować funkcją `deriv()` z pakietu `polynom` opisaną w rozdziale poświęconym wielomianom.

Jako ilustrację wyznaczamy pochodną z funkcji $3(x-2)^2-15$. Zauważmy, że wynik poniższego wyrażenia jest klasy `call`.

```
D(expression(3*(x-2)^2-15), "x")
## 3 * (2 * (x - 2))
```

Policzmy pochodną tej funkcji z użyciem `deriv()`.

```
(wyraz <- deriv(~(x-2)^2+15,"x"))  # wynik jest klasy 'expression'
## expression({
##     .expr1 <- x - 2
##     .value <- .expr1^2 + 15
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- 2 * .expr1
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

Policzmy wartość tej pochodnej w punktach x=1, 2 (tu się zeruje) i 3.

```
x <- 1:3
eval(wyraz)
## [1] 16 15 16
## attr(,"gradient")
##       x
## [1,] -2
## [2,]  0
## [3,]  2
```

Do całkowania można wykorzystać funkcję `integrate()`. Wyznacza ona numerycznie całkę na zadanym, niekoniecznie skończonym, przedziale.

Policzmy więc całkę z gęstości rozkładu normalnego na odcinku $[0, \infty]$.

```
integrate(dnorm, 0, Inf)
## 0.5 with absolute error < 4.7e-05
```

Policzmy całkę z odrobinę bardziej skomplikowanej funkcji.

```
integrate(function(x) sin(x)^2, 0, 600)
## 300.0221 with absolute error < 0.0087
```

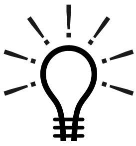
Na półprostej ta całka nie istnieje.

```
integrate(function(x) sin(x)^2, 0, Inf)
## Error in integrate(function(x) sin(x)^2, 0, Inf) :
##   the integral is probably divergent
```

W programie R dostępne są też funkcje do rozwiązywania równań różniczkowych. Więcej informacji można znaleźć w plikach pomocy dla funkcji `odesolve::rk4()` i `odesolve::lsoda()`.

## 2.4 Zapisywanie i odczytywanie danych

Aby wykonać analizę danych musimy te dane najpierw wczytać. Mając wyniki analiz, chcemy je zapisać. W programie R możemy łatwo zapisywać i odczytywać dane z plików tekstowych, plików w formatach binarnych, schowka systemowego, baz danych itp.



Bardzo szczegółowy opis możliwości importu danych z i eksportu danych do progrmu R można znaleźć pod adresem [14]. W tym dokumencie znajduje się wiele ciekawych informacji, o których tu nawet nie wspominamy, np. jak korzystać ze spakowanych zbiorów danych, jak łączyć się z bazami danych, z innymi pakietami statystycznymi, z innymi językami oprogramowania itp.

### 2.4.1 Zapisywanie i odczytywanie danych z plików

Podstawowe funkcje do operacji na plikach i katalogach zostały już przedstawione w podrozdziale 1.6.6. Między innymi przedstawiliśmy tam przykłady prostego użycia funkcji `write.table()` i `read.table()`. Obie te funkcje mają znaczną liczbę dodatkowych argumentów. O tych argumentach i o innych funkcjach do odczytu danych napiszemy poniżej.

#### 2.4.1.1 Funkcja read.table()

Funkcja `read.table()` wczytuje dane o strukturze ramki danych, kolumny odpowiadają kolejnym zmiennym a wiersze kolejnym obserwacjom (przypadkom). Wartości w tej samej kolumnie muszą być tego samego typu, ale typy w różnych kolumnach mogą się różnić. Deklaracja funkcji `read.table()` jest następująca:

```
read.table(file, header=FALSE, sep="", quote="\"'", dec=".",
        row.names, col.names, as.is=!stringsAsFactors,
        na.strings="NA", colClasses=NA, nrows=-1, skip=0,
        check.names=T, fill=!blank.lines.skip, strip.white=F,
        blank.lines.skip=T, comment.char="#", allowEscapes=F,
        flush=FALSE, stringsAsFactors, encoding = "unknown")
```

W pakiecie `utils` znajdują się również cztery inne funkcje o takim samym działaniu, ale innej parametryzacji argumentów domyślnych.

```
read.csv(file, header=TRUE, sep=",", quote="\"", dec=".",
         fill=TRUE, comment.char="", ...)
read.csv2(file, header=TRUE, sep=";", quote="\"", dec=",",
         fill=TRUE, comment.char="", ...)
read.delim(file, header=TRUE, sep="\t", quote="\"", dec=".",
         fill=TRUE, comment.char="", ...)
read.delim2(file, header=TRUE, sep="\t", quote="\"", dec=",",
         fill=TRUE, comment.char="", ...)
```

Jeżeli dane zapisaliśmy w polskiej lokalizacji programu Excel, to po zapisaniu ich w formacie `.csv` separatorem kolumn będzie średnik, a kropką dziesiętną będzie przecinek. W tym przypadku najodpowiedniejszą funkcją do odczytu takich danych jest funkcja `read.csv2()`. Jeżeli korzystamy z angielskiej wersji Excela, to kropką dziesiętną jest kropka, a najodpowiedniejszą funkcją do odczytu tych danych jest funkcja `read.csv()`.

W tabeli 2.11 przedstawiamy opis argumentów funkcji `read.table()`, angielską wersję tych opisów, znaleźć można w pliku pomocy do funkcji `read.table()`. Jak widzimy argumentów jest wiele, warto choć raz o wszystkich przeczytać.

**Tabela 2.11:** Argumenty funkcji `read.table()`

| Argument | Opis |
|---|---|
| `file` | Pole o wartości typu znakowego. Jest to jedyny argument obligatoryjny, tzn. musi być jawnie wskazany. Jego wartość, to ścieżka wskazująca plik, w którym znajdują się dane. Można wskazać plik na dysku lokalnym lub z Internetu, podając adres URL określający, na którym serwerze i w jakim katalogu ten plik się znajduje. To pole może również określać połączenie lub gniazdo (ang. *socket*). Jeżeli zamiast nazwy pliku podamy wartość `"clipboard"`, to dane będą czytane ze schowka systemowego, co jest wygodnym sposobem przenoszenia danych z różnych „office'ów". |
| `header` | Pole o wartości typu logicznego. Jeżeli wartość w tym polu wynosi `TRUE`, to pierwszy wiersz będzie traktowany jako lista nazw kolumn. Jeżeli ten argument nie zostanie podany, a w pierwszej linii w pliku jest o jedno pole mniej pól niż w kolejnych liniach, to R automatycznie potraktuje tę linię jako nagłówek. |
| `sep` | Pole o wartości typu znakowego. Wskazany łańcuch znaków będzie traktowany jako separator kolejnych pól. Wartość `""` (domyślna) powoduje, że każdy biały znak (spacja lub ich ciąg, tabulator, znak nowej linii) jest traktowany jako separator. |
| `quote` | Pole o wartości typu znakowego. Każdy znak występujący w tym łańcuchu jest traktowany jako znak cytowania. Tekst występujący pomiędzy dwoma znakami cytowania traktowany jest jako jedno pole, nawet, jeżeli występują w nim separatory. |
| `dec` | Pole o wartości typu znakowego. Określa jaki znak reprezentuje kropkę dziesiętną. Jeżeli dane zapisane są zgodnie z polskimi standardami, to kropką dziesiętną jest znak `","` (przecinek). |
| `row.names` | Pole o wartości typu wektor napisów lub pojedyncza liczba. Jeżeli jest to wektor łańcuchów znaków, to będzie on traktowany jako nazwy kolejnych wierszy. Jeżeli tym argumentem jest liczba, to jest ona traktowana jako numer kolumny w odczytywanym pliku, która zawiera nazwy wierszy. Jeżeli dane zawierają nagłówek (a więc parametr `header=TRUE`) i pierwsza linia ma o jedno pole mniej niż pozostałe, to pierwsza kolumna automatycznie traktowana jest jako wektor z nazwami wierszy. |
| `col.names` | Wektor napisów. Tym argumentem można określić nazwy kolumn. Jeżeli ten argument nie jest podany, a w pliku nie ma nagłówka, to nazwy kolejnych zmiennych konstruowane są przez złączenie litery "V" z numerem kolumny. |

| | |
|---|---|
| `nrows` | Wartość typu liczbowego. Określa jaka maksymalna liczba wierszy ma być odczytana. |
| `skip` | Wartość typu liczbowego. Określa, ile początkowych linii pliku ma być pominięte, przed rozpoczęciem czytania danych. |
| `check.names` | Wartość typu logicznego. Czy wymuszać sprawdzanie, czy nazwy zmiennych są poprawne, bez niedozwolonych znaków i powtórzeń. |
| `as.is` | Wektor zmiennych typu logicznego lub liczbowego. Domyślnie, każde pole, które nie jest konwertowane na liczbę rzeczywistą lub wartość logiczną, jest konwertowane na zmienną typu `factor`. Jeżeli parametr `as.is` jest wektorem wartości logicznych, to jest on traktowany jako wskazanie, które kolumny mogą być konwertowane na typ `factor`, a które nie. Wektor liczb jest traktowany jako indeksy kolumn, które nie powinny być konwertowane. |
| `na.strings` | Wektor napisów. Wskazuje jakie wartości mają być traktowane jako `NA`, czyli jakie wartości określają brakujące obserwacje. Dodatkowo, jako brakujące obserwacje oznaczane są też puste pola w kolumnach o wartościach typu liczbowego lub logicznego. |
| `colClasses` | Wektor znaków. Każdy znak określa typ kolejnej zmiennej (kolumny). Dopuszczalne wartości to:<br>• NA - domyślna wartość, oznacza automatyczną konwersje,<br>• NULL - ta kolumna będzie pominięta,<br>• jedna z klas atomowych (logical, integer, numeric, complex, character, raw, factor, Date lub POSIXct),<br>• nazwa funkcji konwertującą dany napis na obiekt odpowiedniej klasy (przydatne, jeżeli korzystamy z własnych klas lub odczytujemy dane o specyficznym formacie). |
| `fill` | Wartość typu logicznego. Jeżeli kolejne linie mają różną liczbę wartości, a argument `fill=FALSE` (domyślnie) to funkcja `read.table()` zakończy się błędem. Jeżeli argument `fill=TRUE`, to do krótszych wierszy dodawane będą na koniec wartości `NA`, tak aby uzupełnić wiersze do równej długości. |
| `blank.lines.skip` | Wartość typu logicznego. Jeżeli wynosi `TRUE`, to przy wczytywaniu danych pominięte będą puste linie. |
| `comment.char` | Wartość typu znakowego. Ten znak traktowany jest jako znak komentarza. Po wystąpieniu tego znaku zawartość do końca linii jest ignorowana. |
| `allowEscapes` | Wartość typu logicznego. Jeżeli jest równa `TRUE`, to tekst odczytywany jest zgodnie z zasadami eskejpowania znaków specjalnych. Przykładowo napisy *t* lub *n* są zamieniane na odpowiednio znak tabulacji oraz znak nowej linii. Wartość `FALSE` (domyślna) powoduje, że tekst czytany jest dosłownie, bez żadnych interpretacji. |
| `stringsAsFactors` | Wartość typu logicznego. Jeżeli jest równa `TRUE`, to łańcuchy znaków będą konwertowane na typ wyliczeniowy `factor`. Jeżeli `FALSE`, to będą reprezentowane jako łańcuchy znaków. Domyślna wartość to `TRUE`. |

Przykłady wywołań tej funkcji znaleźć można w rozdziale 1.6.6. W poniższym przykładzie przedstawiamy polecenie odczytujące dane w formacie csv, polecenie odczytujące dane ze schowka systemowego (np. wkopiowane tam z arkusza kalkulacyjnego) oraz polecenie odczytujące dane z pliku tekstowego umieszczonego w Internecie.

Poniżej odczytujemy dane tabelaryczne z plików tekstowych z dysku, ze schowka systemowego i z Internetu. Dane są w różnych formatach, rozdzielane przecinkiem (pierwsza linia), z napisem brak oznaczającym wartości brakujące (druga linia) i być może z niepełnymi wierszami (trzecia linia).

```
dane    <- read.csv2("c:/tajneDane/raportyKGB.csv", sep=",")
listaZakupow <- read.delim("clipboard", header=FALSE, na.strings="brak")
literki <- read.table("http://www.biecek.pl/R/literki.csv", fill=TRUE)
```

#### 2.4.1.2 Funkcja write.table()

Funkcja `write.table()` zapisuje wartość zmiennej (atomowej, wektora, macierzy lub ramki danych) w określonym formatowaniu do pliku tekstowego. Pełna deklaracja tej funkcji jest następująca:

```
write.table(x, file="", append=FALSE, quote=TRUE, sep=" ",
            eol="\n", na="NA", dec=".", row.names=TRUE,
            col.names=TRUE, qmethod=c("escape","double"))
```

Podobnie jak w przypadku funkcji `read.table()` w pakiecie `utils` dostępne są dwie funkcje o identycznym działaniu co `write.table()`, ale przy innej parametryzacji argumentów domyślnych. Obie funkcje przeznaczone są do zapisywania danych w formacie `.csv`.

```
write.csv(x, file = "", sep = ",", dec=".", ...)
write.csv2(x, file = "", sep = ";", dec=",", ...)
```

W tabeli 2.12 przedstawiamy opis wszystkich parametrów funkcji `write.table()`, opis został przygotowany na podstawie pomocy dla tej funkcji (patrz `?write.table`). Poniżej na kilku przykładach pokażemy jak zapisać wektor, liczbę lub ramkę danych do pliku.
Ta instrukcja wypisze wektor liczb na ekranie zamiast do pliku.

```
write.table(runif(100),"")
```

Ta instrukcja wpisze wektor do schowka systemowego.

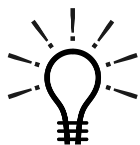
```
write.table(1:100,"clipboard",col.names=FALSE, row.names=FALSE)
```

Ta instrukcja wpisze napis do pliku tekstowego.

```
write.table("poniedzialek","c:/najgorszyDzienWTygodniu.txt")
```

Ta instrukcja zapisze dane do pliku w formacie csv.

```
write.csv2(iris, fill="doExcela.csv")
```



Zapisywanie dużych obiektów typu `data.frame` do pliku w formacie tekstowym może być czasochłonne. Jest to spowodowane przełączaniem się ze stylem formatowania dla każdej kolejnej kolumny, która może być innego typu. Program R sprawdza typ dla każdej kolumny i każdego wiersza formatując wartości w zależności od ich typu.
Jeżeli chcemy szybko zapisać ramkę danych, w której wszystkie wartości są tego samego typu, to warto ją wcześniej przekonwertować na macierz funkcją `as.matrix()`.

**Tabela 2.12:** Argumenty funkcji `write.table()`

| | |
|---|---|
| `x` | Obiekt do zapisania. Pole obligatoryjne. Ten obiekt to najczęściej macierz lub obiekt typu `data.frame`. |
| `file` | Wartość typu znakowego. Jego wartość to ścieżka do pliku, w którym mają być zapisane dane. To pole może określać również połączenie lub gniazdo, tak jak w przypadku funkcji `read.table()`. Podobnie, jeżeli zamiast nazwy pliku podamy wartość `"clipboard"`, to dane będą zapisane do schowka systemowego. Wartość `""` spowoduje wyświetlenie obiektu na konsoli. |
| `qmethod` | Wartość typu znakowego. Wskazuje jak znaki specjalne mają być eskejpowane. Wartość `"escape"` oznacza eskejpowanie w stylu C (z dodaniem backslasha), wartość `"double"` oznacza podwójne eskejpowanie. |
| `sep` | Wartość typu znakowego. Wskazany łańcuch znaków będzie traktowany jako separator kolejnych pól (separatorem nie musi być pojedynczy znak). |
| `eol` | Wartość typu znakowego. Ten łańcuch będzie wstawiany jako koniec linii. W przypadku systemu Windows domyślne zakończenie linii to `eol="\r\n"`. |
| `quote` | Wartość typu logicznego. Wartość `TRUE` powoduje, że pola typu `factor` oraz łańcuchy znaków będą otoczone znakiem `'"'` (cudzysłów). |
| `dec` | Wartość typu znakowego. To pole określa, jaki znak ma reprezentować kropkę dziesiętną, domyślnie jest to `"."` (kropka). Jeżeli dane mają być zapisane zgodnie z polskimi standardami, to kropką dziesiętną powinien być znak `","` (przecinek). |
| `row.names` | Wektor napisów lub wartość typu logicznego. Jeżeli jest to wektor napisów, to będzie on traktowany jak nazwy kolejnych wierszy. Jeżeli jest to wartość logiczna, to będzie ona określała, czy nazwy wierszy powinny być zapisane, czy nie. |
| `col.names` | Wektor napisów lub wartość typu logicznego. Jeżeli jest to wektor napisów, to będzie on traktowany jak nazwy kolejnych kolumn. Jeżeli jest wartość logiczna, to będzie ona określała, czy nazwy kolumn powinny być zapisane czy nie. |

#### 2.4.1.3 Inne funkcje odczytu i zapisu plików tekstowych

Bardziej elastyczną funkcją do odczytywania danych (niekoniecznie tabelarycznych) jest funkcja `scan(utils)`. Dane nie muszą być w postaci tabelarycznej, w każdym wierszu może być różna liczba wartości. Wynikiem funkcji `scan()` jest wektor odczytanych wartości.

```
wektor.wartosci <- scan("nazwa.pliku")
```

Funkcja `scan()` ma równie wiele parametrów co `read.table()`. Najprzydatniejsze z nich to `what` (określający typ odczytywanych wartości), `sep` określający separator dla kolejnych wartości oraz `nmax` umożliwiający określenie maksymalnej liczby wartości do odczytania. Poniższy przykład pozwala odczytać wektor łańcuchów znaków rozdzielonych przecinkami, bez ograniczeń na maksymalną liczbę odczytanych wartości.

```
wektor.lancuchow <- scan("nazwa.pliku", what="character", sep=",")
```

Domyślnie dane odczytywane są ze standardowego strumienia wejściowego, czyli bezpośrednio z konsoli (domyślna wartość parametru `file = ""`). Dzięki temu możemy w skryptach R, wraz z kodem źródłowym umieszczać nieduże zbiory danych. Poniższy skrypt inicjuje macierz `mat` wskazanymi 20 liczbami.
Funkcja `scan()` umożliwia wprowadzanie wartości bezpośrednio z konsoli. Kopiując poniższy kod do programu R, do wektora dane zostanie wprowadzony wektor dwudziestu liczb. Dane wczytane przez funkcję `scan()` są odczytywane jako wektor, poniższą instrukcją zamieniamy je na macierz.

Korzystając z funkcji `scan()` możemy również wprowadzać dane z konsoli. Jeżeli nie podamy żadnego argumentu dla tej funkcji, to R interaktywnie będzie pytał nas o podanie kolejnych wartości, wprowadzanie kończy się gdy podamy pustą linię.

```
dane <- scan()
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4

## Read 20 items
mat <- matrix(dane, 4, 5, byrow=TRUE)
mat[1,]
## [1] 1 2 3 4 1
```

Jeżeli dane zapisane są w pliku tekstowym w blokach o stałych szerokościach pól, to można je odczytywać funkcją `read.fwf()` (gdzie `fwf` to skrót od ang. *fixed width field*). Pierwszy argument tej funkcji (argument `file`) wskazuje nazwę pliku do odczytania, drugi (argument `widths`) jest specyfikacją szerokości kolejnych pól. Tę specyfikację możemy zadać jedną dla całego pliku lub osobną dla każdej linii.

Poniżej dwa przykłady wywołania funkcji `read.fwf()`. W zależności od podanych parametrów, odczytamy z tego samego pliku różne ramki danych. W pierwszym przykładzie specyfikujemy jeden wzorzec dla całego pliku, w drugim osobny wzorzec dla każdej z linii. Załóżmy, że pracujemy z plikiem tekstowym `daneFWF.txt` o następującej zawartości.

```
110 ALA STOP13
111 OLA STOP 5
```

Podajemy format szerokości kolejnych kolumn uniwersalny dla wszystkich linii.

```
(dane <- read.fwf("http://www.biecek.pl/R/dane/daneFWF.txt",
               widths=c(1,2,4,5,2)))
##   V1 V2   V3    V4 V5
## 1  1 10  ALA  STOP 13
## 2  1 11  OLA  STOP  5
```

Podajemy format dla każdej linii osobno.

```
(dane <- read.fwf("http://www.biecek.pl/R/dane/daneFWF.txt",
                widths=list(c(1,2,11), c(2,1,11))))
##   V1 V2          V3 V4 V5          V6
## 1  1 10  ALA STOP13 11  1  OLA STOP 5
```

#### 2.4.1.4 Import i eksport danych w formacie programu Excel

Z Excela dane możemy importować na wiele sposobów. Najprostszym sposobem jest zapisanie danych w Excelu w formacie `csv`, a następnie odczytanie ich w programie R funkcją `read.csv()`.

```
macierz <- read.csv(file="nazwa.pliku.csv", dec=",")
write.csv(macierz, file="nazwa.pliku.csv", dec=",")
```

Należy pamiętać, że Excel z polską lokalizacją stosuje `","` jako kropkę dziesiętną (tak jest też w niektórych innych krajach Europy wschodniej, zdecydowana większość krajów za kropkę dziesiętną wybrało znak kropki).

Ograniczeniem tego sposobu jest problem z polskimi znakami, które mogą być źle zakodowane w pliku `csv`. Drugim problemem jest to, że do pliku `csv` zapisać można tylko jedną zakładkę (ang. *sheet*) z Excela. Jeżeli w pliku Excelowym mamy dużo zakładek, to ten sposób będzie bardzo nieefektywny.

Innym sposobem importowania danych jest przekazywanie ich poprzez schowek. W Excelu zaznaczamy i kopiujemy do schowka wybrane komórki, a w programie R odczytujemy je funkcją `read.table()` (patrz kolejny podrozdział). Jeszcze innym sposobem jest skorzystanie z interfejsu RODBC. Poniższy sposób odczytywania danych działa z plikami Excela oraz Accessa. Odczytajmy dla przykładu wszystkie dane z pliku `plik.xls` z zakładki `Zakladka1`. O interfejsie ODBC jeszcze powiemy przy okazji baz danych. Dodajmy tutaj, że korzystając z tego interfejsu możemy nie tylko odczytywać dane, ale również dodawać/usuwać zakładki oraz dane w plikach Excela.

Łączymy się z plikiem Excela jak z bazą danych. Dla nowszych formatów plików Excela możemy też użyć funkcji `odbcConnectExcel2007()`.

```
library("RODBC")
plikXLS <- odbcConnectExcel("plik.xls")
sqlQuery(plikXLS, "select * from \"Zakladka1\"")
```

Jeszcze inne sposób odczytania danych z formatu Excela to użycie funkcji `read.xls()` z pakietu `xlsReadWrite` lub z pakietu `gdata`. Obie odczytują dane z wybranej zakładki, wskazanej przez argument `sheet`, używają do tego różnych mechanizmów. Funkcja `read.xls()` z pakietu `xlsReadWrite` korzysta z zewnętrznej biblioteki napisanej w Delphi, która pozwala na odczytywanie danych z binar-

nego formatu Excela. Aby ta funkcja działała wymagane jest zainstalowanie też tej biblioteki. Sama biblioteka nie jest na licencji GPL, więc należy uważać na jej licencję jeżeli chcemy wykorzystać tę funkcję w komercyjnych zastosowaniach.

Funkcja `read.xls()` z pakietu `gdata` używa skryptu perlowego do konwersji danych binarnych Excela do pliku `csv` a następnie wczytuje dane z użyciem standardowej funkcji `read.table()`. Wadą takiego rozwiązania jest to, że wymaga zainstalowanego interpretera Perla i że konwersja do `csv` psuje informacje o kodowaniu, polskie literki mogą się źle odczytać.

Poniżej prezentujemy przykład użycia obu funkcji. Aby mieć pewność, że wywołana jest funkcja z właściwego pakietu, używamy operatora zasięgu :: wskazując jawnie pakiety z których wywołujemy funkcje. Jeżeli wczytany jest tylko jeden z tych pakietów nie trzeba go jawnie wskazywać. Jeżeli wczytane są oba pakiety, to domyślnie najpierw funkcja jest szukana w ostatnio czytanym pakiecie.

```
library(xlsReadWrite)
dane <- xlsReadWrite::read.xls("dane.xls", sheet = 1, from=2, type="
    character")
```

W przypadku pakietu `gdata` wygląda to bardzo podobnie.

```
library(gdata)
dane <- gdata::read.xls("dane.xls", sheet = 1)
```

#### 2.4.1.5 Import i eksport danych z użyciem schowka

Dostęp do schowka uzyskuje się podając `"clipboard"` zamiast nawy pliku w funkcji do odczytu lub zapisu danych. Tak więc instrukcja:

```
wektor <- scan("clipboard")
```

odczyta dane ze schowka i zapisze do zmiennej `wektor`, a polecenie:

```
write.csv(macierz, "clipboard")
```

zachowa w schowku wartości zmiennej `macierz` w formacie tabelarycznym. Różne programy i pakiety statystyczne zapisując i odczytując dane mogą używać różnego formatowania, np. różnych separatorów. Dlatego, jeżeli dane nie wczytają się poprawnie, to być może należy użyć innej parametryzacji.

#### 2.4.1.6 Import danych z SPSS'a

Jeżeli dane są zapisane w formacie programu SPSS, to aby je wczytać potrzebny jest pakiet `Hmisc`. W pakiecie `Hmisc` znajduje się wiele funkcji do importu danych zapisanych w plikach binarnych w formacie różnych programów. Do importu formatu SPSS służy funkcja `Hmisc::spss.get()`. Poniżej przykładowa sesja.
Odczytujemy i wyświetlamy dane z formatu programu SPSS.

```
library(Hmisc)
dane  <- spss.get("http://www.biecek.pl/R/dane/daneSPSS.sav")
dane2 <- spss.get("http://www.biecek.pl/R/dane/daneSPSS.sav", datevars="
    URODZONY")
head(dane2, 4)
```

```
##   WIEK WAGA DIAGNOZA   URODZONY
## 1   18   62 zdrowy   1989-10-07
## 2   19   66 zdrowy   1988-01-01
## 3   21   98 chory    1986-11-06
## 4   28   74 chory    1979-12-01
```

W wyniku wykonania tego polecenia do ramki danych zostaną wczytane dane (dokładniej informacje z widoku `Data View` programy SPSS). Aby mieć dostęp do danych z widoku `Variable View` należy użyć atrybutów poszczególnych kolumn. Przykładowo, aby odczytać informacje o opisie danej kolumny (pole `Label` w widoku `Variable View`) możemy wykonać następujące polecenia:
Opis kolumny WIEK w ramce danych.

```
attributes(dane$WIEK)$label
##                          WIEK
## "Wiek osoby w chwili badania"
```

Jak wyżej, tyle że funkcją label z pakietu `Hmisc`.

```
label(dane$WIEK)
##                          WIEK
## "Wiek osoby w chwili badania"
```

Rozbudowane podsumowanie zmiennej `WIEK`.

```
describe(dane$WIEK)
## dane$WIEK : Wiek osoby w chwili badania
##       n missing  unique    Mean
##      11       0       9   22.09
##
##           16 18 19 20 21 24 25 27 28
## Frequency  1  1  1  1  2  2  1  1  1
## %          9  9  9  9 18 18  9  9  9
```

Wiele niskopoziomowych funkcji do odczytu plików w formatach innych programów, znajduje się w pakiecie `foreign`. Jest tam np. funkcja `read.spss()`, również odczytująca dane z formatu SPSS. Funkcje z pakietu `foreign` są wykorzystywane przez omawiany w tym rozdziale pakiet `Hmisc`.

#### 2.4.1.7 Import danych z programu Matlab

Pakiety programu R związane z programem Matlab to `matlab` (emulator Matlaba) i `R.matlab` (pakiet do komunikacji pomiędzy programem R a Matlabem, również do odczytywania i zapisywania plików w formacie MAT). Szczegółowej dokumentacji tych pakietów należy szukać na stronach CRAN. Poniżej podamy przykład jak odczytać pliki zapisane w formacie MAT. Wykorzystamy do tego funkcję `R.matlab::readMat()`. Odczytuje ona pliki MAT zapisane w wersjach V4, V5 i V6. Dane są odczytywane jako lista, której kolejne pola odpowiadają kolejnym zmiennym zapisanym w pliku MAT.
Odczytujemy i wyświetlamy strukturę pliku w formacie Matlaba.

```
library(R.matlab)
str(daneMat <- readMat("http://www.biecek.pl/R/dane/daneMatlab.MAT"))
```

```
## List of 2
##  $ normalny   : num [1:10, 1:10]  0.1352 -0.1390 -1.1634  1.1837
    -0.0154 ...
##  $ wykladniczy: num [1:10, 1:10] 0.540 0.859 0.663 1.097 0.837 ...
##  - attr(*, "header")=List of 3
##   ..$ description: chr "MATLAB 5.0 MAT-file, Platform: PCWIN, Created
    on: Sat Nov 03 12:59:22 2007 "
##   ..$ version    : chr "5"
##   ..$ endian     : chr "little"
```

W tym pliku zapisane są zmienne o nazwach `normalny` i `wykladniczy`.

```
dim(daneMat$normalny)
## [1] 10 10
```

Osoby korzystające z Matlaba z pewnością zainteresują się interfejsem `RMatlab`, pozwalającym na wywoływanie komend R z poziomu Matlaba i komend Matlaba z poziomu R. Więcej informacji na ten temat można znaleźć w [15].

#### 2.4.1.8 Import danych z SAS

Jest kilka sposobów na odczytanie danych zapisanych w formacie SAS. Najłatwiejszym jest skorzystanie z funkcji `foreign::read.ssd()`, np. w następujący sposób:

```
library(foreign)
dane <- read.ssd("http://www.biecek.pl/R/dane","daneSAS.sas7bdat")
```

Minusem tego rozwiązania jest konieczność posiadania zainstalowanego SASa. Funkcja `read.ssd()` uruchamia skrypt SASa, korzystający z procedury `PROC COPY` konwertującej dane na bardziej przenośny format, który następnie jest wczytywany do programu R.

Inne możliwe sposoby odczytywania danych z SASa to użycie jednej z funkcji `Hmisc::sas.get()`, funkcji `foregin::read.xport()`, lub funkcji `sas7bdat::read.sas7bdat()` (nie wymaga instalacji SASa). Można też wykorzystać instrukcje programu SAS opisujące formatowanie pliku z danymi, tak by wykorzystując SASową definicję tzw. DATA STEP wczytać dane do R. Do tego służą funkcje `SAScii::read.SAScii()` (wczytuje dane) i `SAScii::parse.SAScii()` (parsuje skrypt SAS).

#### 2.4.1.9 Import danych z bibliotek R

Niektóre pakiety poza funkcjami udostępniają również zbiory danych. Dostęp do tych zbiorów uzyskuje się korzystając z funkcji `data()`. Na poniższym przykładzie przedstawiamy odczyt zbioru danych `Coffee` z pakietu `BSDA` (ten zbiór danych zawierają informacje o produktywności pracowników z i bez przerwy na kawę) oraz zbiór danych `cars` z pakietu `stats` (dane zawierają informacje o średniej prędkości w milach na godzinę na odcinkach o różnych długościach).

Wszyscy pracownicy wiedzą, że ważniejsze od samego faktu istnienia przerwy jest to, czy wlicza się ją do czasu pracy.

Wczytujemy zbiór danych Coffee z biblioteki BSDA. Wypiszmy na ekranie pierwsze kilka wierszy tego zbioru danych.

```
library(BSDA)
data(Coffee)
head(Coffee)
##   Without With differ sgnrnks
## 1      23   28      5     9.0
## 2      35   38      3     7.0
## 3      29   29      0     0.0
## 4      33   37      4     8.0
## 5      43   42     -1    -3.0
## 6      32   30     -2    -5.5
```

Poniższe komendy pozostawiamy do samodzielnych ćwiczeń.

```
data(cars)
plot(cars)
lines(lowess(cars))
```

Aby wyświetlić listę wszystkich zbiorów danych, które znajdują się we włączonych pakietach, należy wywołać funkcję `data()` bez argumentów.

### 2.4.2 Zapisywanie i odczytywanie danych z baz danych

Program R przechowuje w pamięci wszystkie dane, na których operuje. To rozwiązanie świetnie się sprawdza, jeżeli zbiory na których operujemy nie są bardzo duże. Jednak w pewnych zagadnieniach nie jesteśmy nawet w stanie wczytać do pamięci całego zbioru danych, nie mówiąc już o przetwarzaniu tych danych. W tej sytuacji warto dane przechowywać w bazie danych, a do pamięci wczytywać tylko te fragmenty, na których chcemy wykonać jakąś operację.

Nie ma co się tutaj rozwodzić nad zaletami baz danych (w szczególności będziemy mówić o relacyjnych bazach danych), ponieważ operując na dużych zbiorach danych (dużych, czyli ponad 10 milionów wierszy) nie mamy praktycznie wyboru i musimy z nich korzystać. Nawet dla zbiorów danych o średnich rozmiarach, korzystając z baz danych zyskujemy na możliwości łatwego współdzielenia danych z innymi aplikacjami/użytkownikami. Poniżej przedstawiamy dwa przykłady komunikacji R z bazami danych z wykorzystaniem pakietów `RODBC` i `RMySQL`. Pierwszy z pakietów pozwala na łączenie się z różnymi bazami danych poprzez sterowniki ODBC, drugi pakiet jest dedykowany do korzystania z bazy danych MySQL.

Połączenie ODBC musi być przygotowane/zdefiniowane w systemie. W przykładzie poniżej otwieramy połączenie ze wskazaną bazą danych MySQL. Zobaczmy jakie są tabele w tej bazie danych i odczytajmy dane z tabeli `tabelaA`.

```
library("RODBC")
polaczenie <- odbcDriverConnect("")
polaczenie <- odbcConnect("baza", uid="uzytkownik", case="tolower")
sqlTables(polaczenie)
sqlQuery(polaczenie, "select * from tabelaA")
close(polaczenie)
```

Możemy też połączyć się z bazą danych PostgreSQL (i klonami, jak np. Amazonowy RedShift), używając biblioteki `RPostgreSQL`. Otwieramy połączenie ze wskazaną bazą danych PostgreSQL. Odczytajmy wszystkie wiersze z tabeli `tabelaA`.

```
require(RPostgreSQL)
drv <- dbDriver("PostgreSQL")
db.con <- dbConnect(drv, host="host", dbname="baza", user="uzytkownik",
    password="haslo")
(tmp <- dbGetQuery(db.con, "SELECT * FROM tabelaA"))
```

Podobnie wygląda łączenie z bazą MySQL używając biblioteki `RMySQL`. Otwieramy połączenie ze wskazaną bazą danych MySQL. Odczytajmy wszystkie wiersze z tabeli `tabelaA`.

```
library(RMySQL)
polaczenie <- dbConnect(MySQL(), user="uzytkownik", dbname="baza", host="
    host", password="haslo")
zapyt <- dbSendQuery(polaczenie, "SELECT * FROM tabelaA")
dane  <- fetch(zapyt, n = -1)
```

Excel do wersji 2007 ma ograniczenie na maksymalną liczbę kolumn (max to 256) i wierszy (max to 65536), przez co jest nieprzydatny do poważnych zastosowań.

Używając funkcji `RODBC::odbcConnectAccess()` można połączyć się z Accesową bazą danych a używając funkcji `RODBC::odbcConnectExcel()` można na plikach Excela wykonywać operacje jak na bazie danych.
Łączymy się z bazą Access.

```
library("RODBC")
polaczenie <- odbcConnectAccess("plik.mdb", uid="uzytkownik")
```

#### 2.4.2.1 Inne funkcje do importu danych

W tabeli 2.13 znajduje się lista funkcji umożliwiających import danych zapisanych w formatach innych pakietów statystycznych. Nie ma potrzeby omawiania tych funkcji szczegółowiej, bowiem korzysta się z nich w sposób podobny do funkcji `read.table()`.

**Tabela 2.13:** Lista wybranych funkcji do importu danych z popularnych programów statystycznych. Za wyjątkiem ostatnich trzech funkcji, pozostałe pochodzą z pakietu `foreign`.

| | |
|---|---|
| `read.S()` | Wczytywanie danych w formacie S. |
| `read.arff()` | Wczytywanie danych w formacie Attribute-Relation File Format wykorzystywanym przez program Weka. |
| `read.dbf()` | Wczytywanie danych z pliku bazy danych DBF. |
| `read.dta()` | Wczytywanie danych z pliku w formacie Stata. |
| `read.epiinfo()` | Wczytywanie danych z pliku w formacie Epi (rozszerzenie .REC). |
| `read.mtp()` | Wczytywanie danych z pliku w formacie Minitab. |
| `read.octave()` | Wczytywanie danych z pliku w formacie Octave. |
| `read.spss()` | Wczytywanie danych z pliku w formacie SPSS. |
| `read.ssd()` | Wczytywanie danych z pliku w formacie SAS (przez wygenerowanie programu konwertującego na format rozpoznawalny przez R). |
| `read.systat()` | Wczytywanie danych z pliku w formacie Systat. |
| `read.xport()` | Wczytywanie danych w formacie SAS XPORT. |
| `sas.get()` | Wczytywanie plików w formacie SAS. Funkcja z pakietu `Hmisc`. |
| `spss.get()` | Wczytywanie plików w formacie SPSS. Funkcja z pakietu `Hmisc`. |
| `readMat()` | Wczytywanie plików Matlaba .MAT. Funkcja z pakietu `R.matlab`. |

### 2.4.3 Inne sposoby odczytywania i zapisywania danych

Odczytywać i zapisywać dane możemy nie tylko z plików z danymi w postaci tabelarycznej, ale również z dowolnych plików tekstowych. Poniżej przedstawimy mechanizm odczytywania i zapisywania plików za pomocą połączeń (ang. *connections*). Jest to niskopoziomowy sposób odczytywania plików, podobny do sposobu korzystania z plików w niskopoziomowych językach programowania.

W przypadku używania połączeń, interakcja ze źródłem danych składa się z trzech etapów:

1. Otwarcie połączenia z plikiem. Lista funkcji, które otwierają połączenia znajduje się w tabeli 2.14. Wszystkie te funkcje wymagają podania przynajmniej dwóch argumentów, `description` z opisem połączenia (ścieżka do pliku, nazwa gniazda itp), oraz `open` określający w jakim trybie otwierane jest połączenie. Tryby `"r"`, `"rt"`, `"rb"` to tryby czytania (w tym tekstowego i binarnego), `"w"`, `"wt"`, `"wb"`, to tryby zapisywania do połączenia (odpowiednio tekstowego i binarnego), `"a"`, `"at"`, `"ab"`, to tryby dopisywania do połączenia (do końca pliku), `"r+"`, `"r+b"`, `"w+"`, `"w+b"`, `"a+"`, `"a+b"`, to tryby mieszane do zapisywania i odczytywania.

2. Interakcja z połączeniem. Lista funkcji, które pozwalają na czytanie lub zapisywanie do połączenia znajduje się w tabeli 2.14.

3. Zamknięcie połączenia funkcją `close()`.

Na poniższym przykładzie prześledźmy skrypt użyty do parsowania strony internetowej.
Odczytujemy źródło HTML strony www z informacjami o pracownikach IMPANu, do otwarcia połączenia wykorzystujemy funkcję `url()`. Odczytujemy całą zawartość tej strony i zamykamy połączenie.

```
polacz <- url("http://www.impan.gov.pl/pracownicy.html",open="r")
zawartoscUrla <- readLines(polacz)
close(polacz)
```

Teraz parsujemy odczytane informacje. Wybieramy linie z informacjami o pracownikach. Z tych linii zapisanych w formacie HTML wyciągamy dane pracowników.

```
osoby <- zawartoscUrla[grep("<li class=\"staff\">", zawartoscUrla)]
pracownicy <- sapply(strsplit(osoby, "</?strong>"),function(x) x[2])
```

Wyświetlamy 6 pierwszych nazwisk.

```
head(pracownicy)
## [1] "prof. dr hab. Zofia Adamowicz" "dr hab. Kazimierz Alster"
## [3] "dr Witold Bednorz"             "dr Marek Beska"
## [5] "dr Przemyslaw Biecek"          "dr hab. Adam Bobrowski"
```

Do wyświetlania zawartości plików lub stron internetowych możemy też wykorzystać polecenia `url.show()` i `file.show()`. Wyświetlają one wskazane pliki w oknie programu R. Korzystając z funkcji `download.file()` możemy programem R pobrać wskazany plik z Internetu na dysk lokalny.

Nie wszystkie urządzenia i protokoły są dostępne pod każdym systemem operacyjnym. Aby sprawdzić, czy konkretny protokół lub urządzenie graficzne jest dostępne możemy posłużyć się funkcją `capabilities()`.

Urządzenia oznaczone wartością `TRUE` są dostępne.

```
capabilities()
##     jpeg      png    tcltk      X11 http/ftp  sockets   libxml
##     TRUE     TRUE     TRUE    FALSE     TRUE     TRUE     TRUE
##     fifo   cledit    iconv      NLS  profmem
##    FALSE     TRUE     TRUE     TRUE    FALSE
##
```

**Tabela 2.14:** Lista funkcji z pakietu `base` do operacji na źródłach danych (plikach, kolejkach i gniazdach) i połączeniach

| | |
|---|---|
| `file()` | Do nawiązywania połączenia z plikami na lokalnych dyskach lub w zdalnych katalogach wskazywanych przez adres URL lub z przedefiniowanymi połączeniami `stdin` (odczytywanie danych z konsoli) lub `clipboard` (odczytywanie danych ze schowka). |
| `url()` | Do nawiązywania połączenia z plikami poprzez adres URL. Można korzystać z protokołów `http://`, `ftp://` oraz `file://`. |
| `gzfile()` | Do nawiązywania połączenia z plikami spakowanymi algorytmem gzip. |
| `bzfile()` | Do nawiązywania połączenia z plikami spakowanymi algorytmem bzip2. |
| `unz()` | Do nawiązywania połączenia (wyłącznie w trybie czytanie) z plikiem archiwum w formacie `zip`. |
| `pipe()` | Do nawiązywania połączenia (również z innymi programami) poprzez mechanizm potoków (ang. *pipe*). |
| `fifo()` | Do nawiązywania połączenia poprzez kolejkę FIFO (Windows nie obsługuje kolejek FIFO). |
| `socketConnection()` | Do nawiązywania połączenia przez gniazdo. |
| `seek()` | Pozwala na zmianę aktualnej pozycji w połączeniu. Dzięki temu możemy wybiórczo odczytywać z dowolnych wskazanych miejsc połączenia. |
| `readLines()` | Odczytuje do `n` linii z połączenia traktowanego jako plik tekstowy, domyślnie `n=-1`, czyli odczytywane są wszystkie linie. Dodatkowymi argumentami można określić sposób kodowania znaków. Wynikiem tej funkcji jest wektor napisów. |
| `readBin()` | Odczytuje do `n` rekordów w formacie binarnym z połączenia, domyślnie `n=-1` czyli wszystkie rekordy są odczytywane. |
| `writeLines()` | Zapisuje do połączenia wskazany wektor napisów. |
| `writeBin()` | Zapisuje do połączenia wskazany wektor rekordów w formacie binarnym. |

### 2.4.4 Zapisywanie grafiki do pliku

Program R umożliwia zapisywanie grafiki do pliku w różnych formatach, w tym png, bmp, jpeg, pdf i ps. Jeżeli interesujący nas rysunek (np. wykres) jest już wyświetlony w graficznym oknie R (to okno o nazwie R Graphics), to aby ten wykres zapisać do pliku, należy uaktywnić okno graficzne a następnie z menu wybrać opcję File\Save as. To rozwiązanie ma dwa poważne minusy. Pierwszy to pracochłonność, gdy zapisać do pliku chcemy nie jeden ale sto obrazków. Drugim minusem jest brak okna graficznego gdy R jest uruchomiony w trybie tekstowym.

Grafikę do plików można też zapisywać w sposób zautomatyzowany, korzystając z funkcji z pakietu grDevices takich jak png(), bmp(), jpeg(), pdf(), poscript() czy bitmap(). Funkcje te przekierowują domyślne wyjście dla grafiki do wskazanego pliku.

Funkcja dev.off() powoduje zamknięcie przekierowania i zapisanie wykresu pod wskazaną nazwą.

W poniższym przykładzie zapisujemy wykres do pliku obrazek.png w formacie png i w rozdzielczości 640x480.
Otwieramy plik do zapisu rastrowej grafiki. Wpisujemy do pliku przykładowe wykresy dla funkcji plot(), zapisany będzie tylko jeden, ostatni wykres. Zamykamy i zapisujemy wyniki w pliku obrazek.png.

```
png("obrazek.png", width = 640, height = 480)
example(plot)
dev.off()
```

Funkcje png(), bmp() i jpeg() przyjmują takie same argumenty, tzn. nazwę pliku, do którego zapisana ma być grafika, szerokość i wysokość zapisywanego obrazka, jednostka w której podane są rozmiary obrazka (domyślnie w pikselach) i kolor tła. Funkcja jpeg() ma dodatkowy argument quality określający jakość grafiki, czyli stopień kompresji. Ten argument przyjmować może wartości od 1 do 100, przy czym mniejsze wartości oznaczają większą kompresję, a co za tym idzie większe straty na jakości.

Dla formatów rastrowych domyślnie wymiary podawane są w pikselach, a dla formatów wektorowych (pdf i ps) domyślną jednostką są cale. Poniższy przykład zapisuje ten sam wykres w pliku o formacie pdf. Zapisany rysunek będzie miał wymiary $8 \times 6$ cali.

Otwieramy plik do zapisu wektorowej grafiki. Wpisujemy do pliku przykładowe wykresy dla funkcji plot(), kolejne wykresy będą dostępne jako kolejne strony wynikowego pliku pdf. Zamykamy i zapisujemy wyniki w pliku obrazek.pdf.

```
pdf("obrazek.pdf", width = 8, height = 6)
example(plot)
dev.off()
```

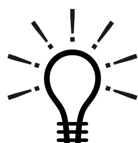
W wersji linuxowej R, do użycia niektórych formatów graficznych wymagane są zainstalowane biblioteki X11. Jeżeli korzystamy z serwera, gdzie takich bibliotek nie ma, to pozostają nam do użycia jednie funkcje bitmap() i postscript().

W programie R wykresy i grafiki wyświetlane są na urządzeniach graficznych. Urządzenie graficzne może być skojarzone z oknem graficznym `R Graphics` w programie R GUI. To okno wyświetla zawartość urządzenia graficznego na ekranie komputera. Możemy funkcją `X11()` otworzyć kolejne urządzenie graficzne i stowarzyszone z nim okno `R Graphics`. W ten sposób możemy w różnych oknach rysować różne wykresy, następnie ustawiając te okna obok siebie możemy łatwo porównać ww. wykresy. Również funkcje `png()`, `bmp()`, `jpeg()`, `pdf()`, `postsript()` powodują otwarcie nowego urządzenia graficznego, przy czym zawartość tego okna nie jest nigdzie wyświetlana. Nowo otwarte urządzenie graficzne staje się urządzeniem domyślnym i wszystkie instrukcje graficzne od tej chwili będą wykonywane na tym urządzeniu. Funkcja `dev.off()` zamyka aktywne urządzenie graficzne i (opcjonalnie) zapisuje zawartość urządzenia graficznego na dysku we wskazanym formacie.
Funkcje umożliwiające otwieranie, zamykanie, czy przełączanie kolejnych urządzeń graficznych są przedstawione w tabeli 2.15.

**Tabela 2.15:** Funkcje z pakietu `grDevices` do zarządzania urządzeniami graficznymi

| | |
|---|---|
| `dev.cur()` | Aktualnie otwarte urządzenie graficzne. |
| `dev.list()` | Lista otwartych urządzeń graficznych. |
| `dev.next(which = dev.cur())` | Uaktywnia kolejne (wskazane) urządzenie. |
| `dev.prev(which = dev.cur())` | Uaktywnia poprzednie (wskazane) urządzenie. |
| `dev.off(which = dev.cur())` | Zamyka wskazane urządzenie graficzne. |
| `dev.set(which = dev.next())` | Uaktywnia wskazane urządzenie graficzne. |

W pakiecie `grDevices` dostępne są też funkcje do zapisu grafiki w mniej popularnych formatach, np. funkcje `xfig()` (zapisuje rysunek w formacje xfig) i `pictex()` (zapisuje instrukcje LaTeX'a generujące dany rysunek). Jeżeli przygotowujemy wykresy dla LaTeXa, to warto zapisać je przy pomocy funkcji `pictex()`. Wykres zapisany w postaci tekstowej często łatwiej zmodyfikować ręcznie, bez generowania go na nowo w programie R, o ile zmiana nie jest duża.



Wykresy wygenerowane przez te same funkcje R mogą różnie wyglądać w zależności od urządzenia graficznego, na którym są rysowane i rozdzielczości, w której są zapisywane. Przykładowo, niektóre czcionki mogą nie być dostępne dla formatu `pdf`, niektóre szerokości linii nie będą dostępne dla formatów `png`, `jpeg`, `bmp` i innych dla grafiki rastrowej (nie będzie możliwe narysowanie linii cieńszej niż jeden punkt, tego ograniczenia nie ma dla formatów wektorowych). Przy domyślnych ustawieniach polskie znaki rysowane są bez problemu w formatach rastrowych, ale nie w formacie `pdf`.Wybór formatu, do którego zapisujemy powinien zależeć od tego, co dalej z tym rysunkiem chcemy robić. Do publikacji grafik w Internecie lepiej nadają się formaty rastrowe (np. `jpeg`, `png`), jeżeli rysunek chcemy wydrukować lub dołączyć do publikacji, to lepszy efekt otrzymamy stosując formaty wektorowe (np. `pdf`, `ps`, `svg`).

## 2.5 Tryb wsadowy

Program R może być uruchomiony w trybie interaktywnym albo trybie wsadowym. W trybie interaktywnym z graficznym interfejsem użytkownika można go uruchomić poleceniem `Rgui.exe` (w systemie Windows). Można też uruchomić R bez interfejsu graficznego poleceniem `R.exe` (w systemie Windows) lub `R` (w pozostałych systemach operacyjnych).

Jeżeli nie potrzebujemy interakcji z R to możemy uruchomić program `Rterm.exe` lub `R.exe` (pod Windows) lub `R` (inne systemy) z parametrem `--vanilla`. Jako strumień wejściowy wskazać można zawartość pliku z listą poleceń programu R do wykonania. Strumień wyjściowy można przekierować do wskazanego pliku. Parametr `vanilla` powoduje, że R po zakończeniu pracy nie pyta się, czy zapamiętać historię poleceń oraz przestrzeń roboczą.

Przykładowe uruchomienie w tym trybie:

```
Rterm --vanilla <plik.wejsciowy.R >plik.wynikowy.txt
```

Po wykonaniu poleceń z pliku wejściowego środowisko R zostanie zamknięte a przestrzeń robocza nie będzie zapisana. Należy pamiętać, aby w wykonywanym skrypcie ręcznie zapisać otrzymane wyniki, np. funkcją `save.image()`. O funkcjach do zapisu wyników w plikach tekstowych przeczytać można w rozdziale 2.4.

Użytkownicy systemu Linux mogą też uruchamiać R w trybie skryptowym. Aby to zrobić potrzebny jest pakiet `littler`. Pod Ubuntu można go zainstalować poleceniem `apt-get install littler`. Po zainstalowaniu tego pakietu dostępny będzie polecenie `r`. Teraz z poziomu powłoki można wykonać dowolną funkcję R. W poniższym przykładzie używamy R jako kalkulatora na poziomie powłoki, bez uruchamiania całego środowiska R.

Nazwa littler oznacza właśnie małą literkę r.

```
cogito@tofesi:~$  r -e 'print(sin(0.6)^4)'
[1] 0.1016469
```

Możemy też wykorzystać inne polecenia powłoki do zasilenia danymi funkcji dostępnych w programie R. W przykładzie poniżej wykorzystywane są polecenia `ls` i `awk` aby wyświetlić listę podkatalogów katalogu `/home/`. Ta lista przekazywana jest do skryptu R, który ją odczytuje i wypisuje na ekranie podsumowanie rozkładu długości nazw podkatalogów.

```
cogito@tofesi:~$  ls -l /home | awk '!/^total/ {print $8}' |
          r -e 'tmp <- readLines(); print(summary(nchar(tmp)))'
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  3.000   5.000   6.000   6.581   7.000  15.000
```

Fragment `ls -l /home`  awk '!/$^{t}otal/print8$'| generuje listę podkatalogów, która jest następnie odczytywana funkcją `readLines()` oraz poddana dalszej obróbce z użyciem funkcji programu R. Używając programu `r` możemy bardzo wygodnie korzystać z całego bogactwa funkcji pakietu R, bez potrzeby ładowania całego środowiska wraz z interaktywnym interfejsem.

## 2.6 Powtarzalne badania, automatyzowane raporty a programowanie objaśniające

Angielski termin *literate programing* (przetłumaczony tutaj jako programowanie objaśniające) został zaproponowany przez Donalda Knutha, autora między innymi języka TEX. Za tym terminem kryje się idea, którą można opisać w trzech punktach (zobacz również artykuł [16]):

1. programy są bezużyteczne bez opisów/dokumentacji,
2. opisy powinny wyjaśniać program, zwykłe komentarze w kodzie nie są wystarczające,
3. przykładowe kody wykorzystane w opisach powinny być wykonywalne i kompletne.

Jedną z implementacji tej idei znaleźć można w pakietach `Sweave` i `knitr`. Pakiety te umożliwiają tworzenie programów/skryptów R wraz z ich opisami. Takie połączenie być wykorzystane zarówno do dokumentowania kodu, do zapewnienia powtarzalności analiz jak i do automatycznego generowania raportów. Poniżej przedstawimy kilka przykładowych zastosowań, więcej informacji i przykładów można znaleźć w pozycjach [16] i [48].

Punktem wyjściowym jest przygotowanie pliku, w którym znajdą się fragmenty kodu w języku R wraz z opisami w języku naturalnym. W procesie kompilacji tego pliku fragmenty kodu R zostaną wykonane, a ich wynik zostanie dodany do opisu kodu. W ten sposób czytelnik ma 100% gwarancji, że rezultat który jest prezentowany w opisie powstał tylko i wyłącznie w wyniku wywołania wymienionych poleceń R. Wykonywany kod R może dynamicznie zmieniać zawartość końcowego dokumentu, co znajduje wiele najróżniejszych zastosowań, np. do automatycznego generowania zadań egzaminacyjnych.

W ostatnich latach programowanie objaśniające znalazło poważne zastosowanie w temacie powtarzalnych badań (ang. *Reproducible Research*). W czasach gdy publikuje się coraz więcej i coraz mniejsza jest kontrola recenzenta nad pracą, coraz ważniejsze jest zapewnienie, że wyniki z naukowych prac można powtórzyć. W tym celu też coraz częściej wykorzystuje się pakiety `Sweave` i `knitr`.

Poniżej przedstawione będą oba pakiety. Pakiet `knitr` uważany jest za następcę `Sweave`. Jest łatwiejszy w użyciu, pozwala na wyprodukowanie zarówno plików w formacie `pdf` jak i `html` oraz w wielu innych formatach (pośrednio dzięki wykorzystaniu programu `pandoc` konwertującego różne formaty). Zaletą pakietu `Sweave` jest większa kontrola nad procesem przetwarzania. Poniżej przedstawimy najpierw proste zastosowanie pakietu `knitr`, następnie `slidify` a następnie więcej o pakiecie `Sweave`.

### 2.6.1 Pakiet `knitr` a raporty w języku markdown / HTML

Cel, który chcemy osiągnąć jest następujący to możliwie proste zintegrowanie kodu R i sformatowanych opisów w języku naturalnym. Do kodu w programie R,

chcemy dodać opis odpowiednio sformatowany wyjaśniający co ten kod robi (niektóre elementy chcielibyśmy podkreślić, wytłuścić, powiększyć). Jednocześnie chcielibyśmy aby z takiej mieszaniny R i języka naturalnego można było wygenerować dokument zawierający zarówno sformatowany opis, jak i kod R jak i wyniki wygenerowane przez ten kod R. Pośrednio umożliwi to również automatyczne generowanie raportów, w których być może chcielibyśmy mieć opisy i wyniki, ale bez wyświetlania kodu R.

Najłatwiej ten cel osiągnąć z pakietem `knitr` [48]. Używając go można definiować raporty przez połączenie języka `markdown` (prosty i łatwy do nauczenia się język) i R. Jest to szczególnie proste gdy korzysta się z edytora `RStudio`, który ma wiele udogodnień dla tworzenia raportów w ten sposób.

Zwyczajowo, pliki w których łączy się kod R i markdown oznacza się rozszerzeniem `.Rmd`. Mając taki plik przetwarza się go następnie używając funkcji `knitr()`. Ta funkcja wykonuje instrukcje programu R (można dołączać również kod w innych językach, np. python) a następnie zarówno instrukcje jak i ich wynik zapisuje z użyciem formatu markdown. Zwyczajowo pliki w formacie markdown oznacza się rozszerzeniem `.md`. W wyniku przetwarzania, plik o takim rozszerzeniu będzie dodany do katalogu roboczego. Następnie plik w formacie markdown jest transformowany do formatu html, domyślnie do pliku z rozszerzeniem `.html`. Jeżeli korzystamy z programu `RStudio`, co gorąco polecam, to w oknie edytora dostępny jest przycisk `Knit HTML` (o ile rozszerzenie pliku to `*.Rmd`). Kliknięcie tego przycisku powoduje przekształcenie pliku `Rmd` do formatu markdown, następnie html, następnie wyświetlenie strony html w przeglądarce.

Połączenie R i LaTeX'a z użyciem pakietu `knitr` zapisuje się zazwyczaj w plikach o rozszerzeniu `.Rnw`.

Format markdown jest „lekki", co oznacza, że wymaga niewielu dodatkowych znaków, by określić formatowanie. Jest to też format często wykorzystywany w stronach wiki, przez co wiele osób potrafi korzystać z niego intuicyjnie. Przedstawimy wybrane elementy tego języka na przykładzie.

Przypuśćmy, że poniższy kod to zawartość pliku `przyklad.Rmd`.

````
Przykładowy skrypt łączący markdown i R
========================================

**Wykres pudełkowy** jest bardzo popularną metodą prezentacji
zmienności pojedynczej zmiennej. Przedstawimy go na przykładzie
zbioru danych _iris_, który ma `r nrow(iris)` wierszy.

```{r fig.width=7, fig.height=6}
summary(iris)
boxplot(Petal.Length~Species, data = iris)
```
````

W wyniku przetworzenia tego pliku przez funkcję `knitr()` wygenerowany będzie plik `przyklad.md` o zawartości przedstawionej poniżej.

````
Przykładowy skrypt łączący markdown i R
=======================================

**Wykres pudełkowy** jest bardzo popularną metodą prezentacji
zmienności pojedynczej zmiennej. Przedstawimy go na przykładzie
zbioru danych _iris_, który ma 150 wierszy.


```r
summary(iris)
```

```
##   Sepal.Length   Sepal.Width    Petal.Length   Petal.Width
##  Min.   :4.30   Min.   :2.00   Min.   :1.00   Min.   :0.1
##  1st Qu.:5.10   1st Qu.:2.80   1st Qu.:1.60   1st Qu.:0.3
##  Median :5.80   Median :3.00   Median :4.35   Median :1.3
##  Mean   :5.84   Mean   :3.06   Mean   :3.76   Mean   :1.2
##  3rd Qu.:6.40   3rd Qu.:3.30   3rd Qu.:5.10   3rd Qu.:1.8
##  Max.   :7.90   Max.   :4.40   Max.   :6.90   Max.   :2.5
##       Species
##  setosa    :50
##  versicolor:50
##  virginica :50
##
##
##
```

```r
boxplot(Petal.Length ~ Species, data = iris)
```

![plot of chunk unnamed-chunk-1](figure/unnamed-chunk-1.png)
````

Zwróćmy uwagę, że znak ` to *grawis* a nie prosty apostrof '. W zależności od klawiatury, znak ` znajduje się pod znakiem `Esc` lub obok lewego znaku `Shift`.

Zauważmy, że w poniższym kodzie nastąpiły dwie zmiany:

- Pierwsza dotyczy „krótkich wstawek”. Instrukcje pomiędzy ogranicznikami ’`r ’ a `’ zostaną wykonane z użyciem języka R (technicznie używana jest nazwa „silnika R”).

  Dlatego fragment `` `r nrow(iris)` `` został przekształcony w pojedynczą liczbę `150`.

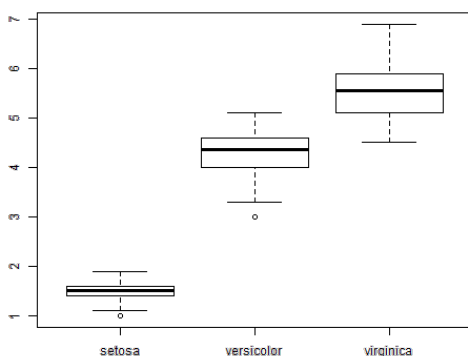
## Przykładowy skrypt łączacy markdown i R

**Wykres pudełkowy** jest bardzo popularna metoda prezentacji zmiennosci pojedynczej zmiennej.
Przedstawimy go na przykladzie zbioru danych *iris*, który ma 150 wierszy.

```
summary(iris)
```

```
##   Sepal.Length   Sepal.Width    Petal.Length   Petal.Width
##  Min.   :4.30   Min.   :2.00   Min.   :1.00   Min.   :0.1
##  1st Qu.:5.10   1st Qu.:2.80   1st Qu.:1.60   1st Qu.:0.3
##  Median :5.80   Median :3.00   Median :4.35   Median :1.3
##  Mean   :5.84   Mean   :3.06   Mean   :3.76   Mean   :1.2
##  3rd Qu.:6.40   3rd Qu.:3.30   3rd Qu.:5.10   3rd Qu.:1.8
##  Max.   :7.90   Max.   :4.40   Max.   :6.90   Max.   :2.5
##        Species
##  setosa    :50
##  versicolor:50
##  virginica :50
##
##
##
```

```
boxplot(Petal.Length ~ Species, data = iris)
```




**Rysunek 2.3:** Strona dokumentu w formacie `html` otrzymana w sposób zautomatyzowany za pomocą pakietu knitr.

- Druga dotyczy „długich wstawek". Instrukcje pomiędzy liniami ` ```{r ...} ` a ` ``` ` zostaną wykonane z użyciem silnika R a w pliku wynikowym pojawią się bloki odpowiadające zarówno wejściowemu kodowi R jak i wynikom. Zarówno wynikom tekstowym (te są poprzedzone znakami ##), jak i wynikom graficznym (te umieszczane są w osobnych plikach).

Łatwo się domyślić co trzeba zrobić by użyty był inny silnik, np. python.

  W linii rozpoczynającej długą wstawkę można dodać dodatkowe parametry. W przykładzie powyżej te parametry określały szerokość i wysokość wygenerowanego wykresu. Takich parametrów wykonania, które można zmieniać, jest znacznie więcej. Np. można wyłączyć raportowanie ostrzeżeń lub błędów. Dokładną listę znaleźć można w pozycji [48]. Najczęściej używane z parametrów to:

- Parametr `results='asis'|'markup'|'hide'`, określa w jakim formacie wynik ma być umieszczony w pliku `md`. Opcja `'markup'` (domyślna) powoduje przekształcenie wyniki do formatu markdown, `'asis'` powoduje wklejenie wyniku z kodu R bez żadnego dodatkowego formatowania, `'hide'` powoduje uruchomienie kodu R, ale nie umieszczanie wyniku w pliku `md`.
- Parametr `tidy=TRUE|FALSE`, określa czy w kod wejściowy powinien być dodatkowo „uporządkowany” przed umieszczeniem go w pliku wynikowym. Wartość `TRUE` dodaje wcięcia, spacje i dba o to by kod wyglądał czytelniej.
- Parametr `highlight=TRUE|FALSE`, określa czy w wynikowym pliku kod R powinien być dodatkowo „kolorowany” (`TRUE`) czy nie (`FALSE`).
- Parametr `dev=pdf|png|...`, określa w jakim formacie mają być przechowywane wykresy wygenerowane przez kod R.
- Parametry `fig.width`, `fig.height`, `dpi`, określają wymiary wykresów (w calach) i rozdzielczość (liczba punktów na cal).
- Parametr `echo=TRUE|FALSE`, określa czy w wynikowym pliku `md` powinien zostać dołączony wejściowy kod R (`TRUE`) czy nie (`FALSE`).
- Parametr `warning=TRUE|FALSE`, określa czy w wynikowym pliku `md` umieszczać komunikaty ostrzeżeń (`TRUE`) czy nie (`FALSE`).

Zarówno krótkich jak i długich wstawek może być w plikach `.Rmd` dowolnie dużo.

W wyniku dalszego przetwarzania, powyższy plik w formacie `md` zostanie przekształcony do formatu `html` przedstawionego na rysunku 2.3. Odpowiednio oznaczone bloki zostały sformatowane. Tekst ponad podkreśleniem złożonym ze znaków ==== został sformatowany jako tytuł, tekst otoczony znakami ** sformatowany został jako tekst pogrubiony, a otoczony znakami _ jako tekst pochylony. Bloki kodu R i kodu wynikowego zostały oznaczone dodatkowym obramowaniem.

### 2.6.2 Pakiet `slidify` a prezentacje w HTML5

Powyżej przedstawione użycie pakietu `knitr` jak i poniżej przedstawione użycie pakietu `Sweave` pozwalają na proste tworzenie dokumentów w formacie `html` czy `pdf`. Zazwyczaj są to raporty lub dokumentacja. Czasem jednak, chcielibyśmy przygotować w podobny sposób prezentację w postaci serii slajdów.

Można do tego celu użyć LaTeXowego pakietu `beamer` lub podobnego, generującego prezentacje. Wymaga to jednak znajomości LaTeXa i jego pakietów. Używając pakietu `slidify` można prosto wygenerować prezentacje w formacie HTML5 używając czystego R. I o tym będzie ten podrozdział.

Pakiet `slidify` nie jest (na dzień dzisiejszy) dostępny w repozytorium CRAN. Aby z niego skorzystać należy wcześniej zainstalować pakiet `devtools` a następnie zainstalować `slidify` z serwera `github`. Aby to zrobić wystarczą trzy poniższe linie.

```
library(devtools)
install_github(c('slidify', 'slidifyLibraries'), 'ramnathv')
library(slidify)
```

Język definiowania prezentacji jest bardzo podobny do języka markdown, o którym pisaliśmy w poprzednikiem podrozdziale. Jedyna istotna różnica jest taka, że w pliku znajdują się separatory dla kolejnych slajdów, zaznaczające gdzie kończy się jeden slajd a zaczyna kolejny. Takim separatorem jest linia złożona z trzech minusów `---` po której występuje pusta linia.

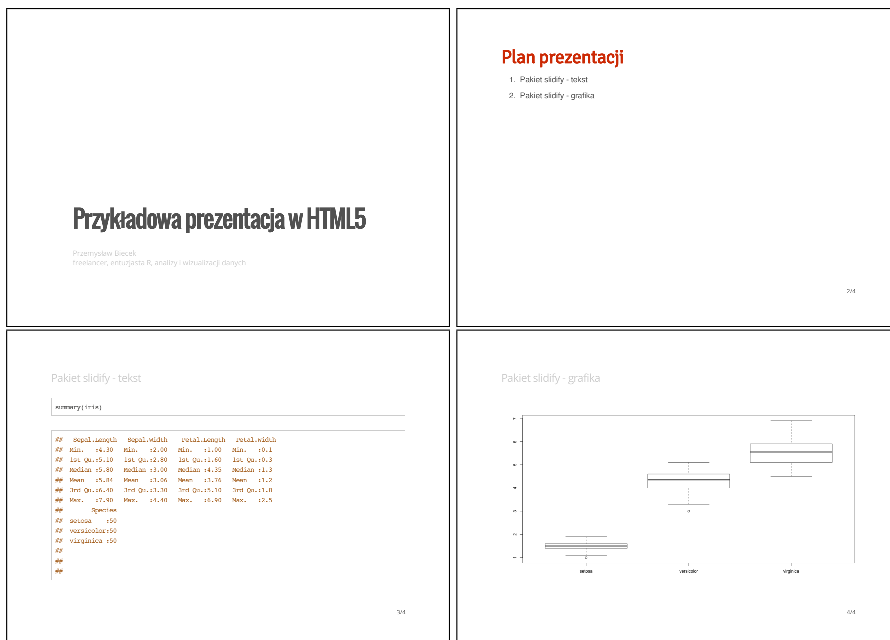
Dodatkowo pierwszy slajd pozwala na określenie szablonu prezentacji (opcja `framework`), sposobu podświetlania kodu (opcja `highlighter`) i innych informacji, takich jak tytuł lub autor prezentacji.

Pracując w programie `RStudio` po utworzeniu pliku o rozszerzeniu `Rmd` z definicją prezentacji, po włączeniu pakietu `slidify`, wystarczy przycisnąć przycisk `Knit HTML` by po kilku sekundach otrzymać czterostronicową prezentację, przedstawioną na rysunku 2.4.

Prezentacje łatwo też opublikować na serwisach `Rpubs.com`, `github.com`.

Poniżej przedstawiony jest kod definiujący te cztery slajdy: tytułowy, ze spisem treści i dwa slajdy integrujący program R i opisy. Z jednej strony mamy proste instrukcje pozwalające na wskazanie nagłówków, treści w języku naturalnym, a z drugiej strony możemy dołączyć wyniki wygenerowane „na żywo” przez program R. Ten kod źródłowy, należy zapisać w pliku z rozszerzeniem `Rmd`. Dynamiczne generowanie prezentacji pozwala na dużą kontrolę spójności wyników. Za każdym razem mamy możliwość sprawdzenia, jaki kod R został wykorzystany do otrzymania prezentowanych wyników. Unika się dzięki temu problemu z pamiętaniem, jaka wersja danych, jaka wersja parametrów doprowadziła akurat do tego konkretnego wyniku.

Pewną wadą pakietu `slidify` jest to, że na dzień dzisiejszy wynik mamy w postaci HTML5. Jeżeli więc przygotowujemy prezentację dla kogoś spodziewającego się formatu `pdf` czy `pptx`, możemy mieć problem. Ostatecznie strony w formacie `html` można „wydrukować” do formatu `pdf`.




**Rysunek 2.4:** Prezentacja wygenerowana z użyciem pakietu `slidify`.

```
---
title       : Przykładowa prezentacja w HTML5
author      : Przemysław Biecek
job        : freelancer, entuzjasta R, analizy i wizualizacji danych
framework   : io2012        # {io2012, html5slides, shower, ...}
highlighter : highlight.js  # {highlight.js, prettify, highlight}
hitheme     : tomorrow      #
widgets     : []            # {mathjax, quiz, bootstrap}
mode        : selfcontained # {standalone, draft}

---
## Plan prezentacji
1. Pakiet slidify - tekst
2. Pakiet slidify - grafika

---
### Pakiet slidify - tekst
```{r}
summary(iris)
```

---
### Pakiet slidify - grafika
```{r fig.width=14, fig.height=7.5, echo=FALSE, warning=FALSE}
boxplot(Petal.Length ~ Species, data = iris)
```
```

### 2.6.3 Pakiet Sweave a raporty w języku LaTeX

Aby ten podrozdział był zrozumiały wymagana jest przynajmniej podstawowa znajomość programu do składu dokumentów LaTeX. Zacznijmy od przykładu. Poniżej przedstawiona jest zawartość pliku `przyklad.Snw`, znajdują się w nim tak komendy języka R jak i LaTeX.

```
\documentclass[a4paper]{article}
\usepackage[cp1250]{inputenc}
\usepackage{Sweave}
\title{\vspace{-3cm}Programowanie objaśniające ze Sweave}
\author{Przemyslaw Biecek}
\begin{document}
\maketitle
```

```
Wykres pudełkowy jest bardzo popularną metodą prezentacji zmienności
pojedynczej zmiennej. Można go wyznaczać również dla kilku zmiennych
(dzięki czemu możemy porównać rozkłady tych zmiennych) lub dla
pojedynczej zmiennej w~rozbiciu na grupy obserwacji.

Kształt tego wykresu jest bardzo charakterystyczny, przypominający
pudełko z~wąsami. Poszczególne elementy wykresu  przedstawiają różne
charakterystyki obserwowanej zmiennej. Środek pudełka przedstawia
medianę, dolna i~górna granica pudełka odpowiada kwartylom z~próby
(odpowiednio dolnemu i~górnemu kwartylowi), kropki przedstawiają
obserwacje odstające. Zakres zmienności (minimum i~maksimum) danej
zmiennej (po ominięciu wartości odstających) zaznaczony jest za
pomocą wąsów wykresu.

<<>>=
dane <- read.csv("http://www.biecek.pl/R/dane/daneSoc.csv",sep=";")
attach(dane)
print(by(wiek, plec, summary))
boxplot(wiek~plec, data = dane, col="black")
@
\begin{center}
<<fig =TRUE , echo =FALSE >>=
boxplot(wiek~plec, data = dane, col="lightgrey")
@
\end{center}
\end{document}
```

Przyjmijmy, że ten plik znajduje się w aktualnym katalogu roboczym programu R. Wykonując polecenie `Sweave("przyklad.Snw")` przetworzymy ten plik wykonując zawarte w nim polecenia programu R. Generujemy kod LaTeX'a z plików Snw.

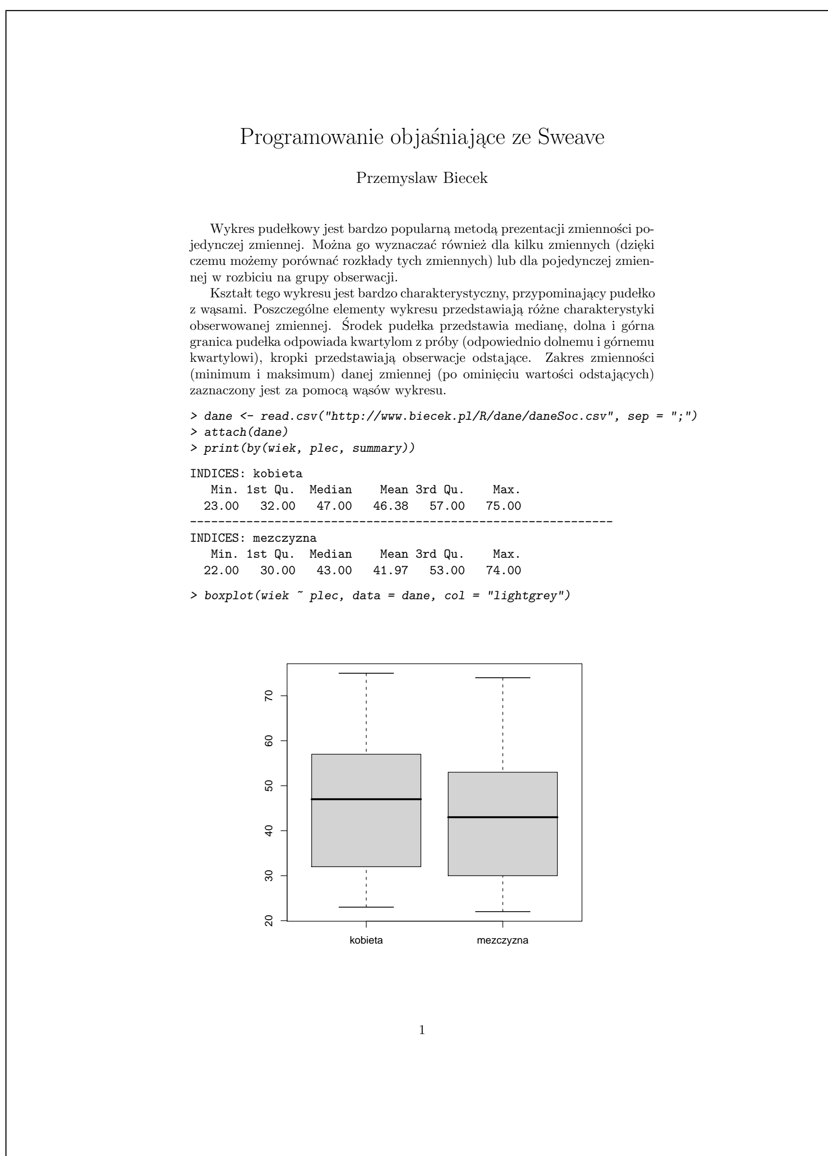
```
Sweave("przyklad.Snw")
## Writing to file przyklad.tex
## Processing code chunks ...
##  1 : echo term verbatim
##  2 : term verbatim eps pdf
##
## You can now run LaTeX on 'przyklad.tex'
```

Otrzymujemy przedstawiony poniżej plik `przyklad.tex` w języku LaTeX. Fragmenty kodu R otoczone znacznikami <<. . .>>= i `@` zostały wykonane. Wyniki zostały wklejone w kod dokumentu LaTeX.

```
\documentclass[a4paper]{article}
\usepackage[cp1250]{inputenc}
\usepackage{Sweave}
\title{\vspace{-3cm}Programowanie objaśniające ze Sweave}
\author{Przemyslaw Biecek}
\begin{document}
\maketitle
Wykres pudełkowy jest bardzo popularną metodą prezentacji zmienności
pojedynczej zmiennej. Można go wyznaczać również dla kilku zmiennych
(dzięki czemu możemy porównać rozkłady tych zmiennych) lub dla
pojedynczej zmiennej w~rozbiciu na grupy obserwacji.

Kształt tego wykresu jest bardzo charakterystyczny, przypominający
pudełko z~wąsami. Poszczególne elementy wykresu  przedstawiają różne
charakterystyki obserwowanej zmiennej. Środek pudełka przedstawia
medianę, dolna i~górna granica pudełka odpowiada kwartylom z~próby
(odpowiednio dolnemu i~górnemu kwartylowi), kropki przedstawiają
obserwacje odstające. Zakres zmienności (minimum i~maksimum) danej
zmiennej (po ominięciu wartości odstających) zaznaczony jest za
pomocą wąsów wykresu.

\begin{Schunk}
\begin{Sinput}
> dane <- read.csv("http://www.biecek.pl/R/dane/daneSoc.csv",sep=";")
> attach(dane)
> print(by(wiek, plec, summary))
\end{Sinput}
\begin{Soutput}
INDICES: kobieta
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  23.00   32.00   47.00   46.38   57.00   75.00
------------------------------------------------------------
INDICES: mezczyzna
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  22.00   30.00   43.00   41.97   53.00   74.00
\end{Soutput}\begin{Sinput}
> boxplot(wiek ~ plec, data = dane, col = "lightgrey")
\end{Sinput}\end{Schunk}
\begin{center}
\includegraphics{przyklad-002}
\end{center}
\end{document}
```



Programowanie objaśniające ze Sweave

Przemyslaw Biecek

Wykres pudełkowy jest bardzo popularną metodą prezentacji zmienności pojedynczej zmiennej. Można go wyznaczać również dla kilku zmiennych (dzięki czemu możemy porównać rozkłady tych zmiennych) lub dla pojedynczej zmiennej w rozbiciu na grupy obserwacji.

Kształt tego wykresu jest bardzo charakterystyczny, przypominający pudełko z wąsami. Poszczególne elementy wykresu przedstawiają różne charakterystyki obserwowanej zmiennej. Środek pudełka przedstawia medianę, dolna i górna granica pudełka odpowiada kwartylom z próby (odpowiednio dolnemu i górnemu kwartylowi), kropki przedstawiają obserwacje odstające. Zakres zmienności (minimum i maksimum) danej zmiennej (po ominięciu wartości odstających) zaznaczony jest za pomocą wąsów wykresu.

```
> dane <- read.csv("http://www.biecek.pl/R/dane/daneSoc.csv", sep = ";")
> attach(dane)
> print(by(wiek, plec, summary))

INDICES: kobieta
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  23.00   32.00   47.00   46.38   57.00   75.00
------------------------------------------------------------
INDICES: mezczyzna
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  22.00   30.00   43.00   41.97   53.00   74.00

> boxplot(wiek ~ plec, data = dane, col = "lightgrey")
```



1

**Rysunek 2.5:** Strona dokumentu w formacie pdf otrzymana w sposób zautomatyzowany za pomocą pakietu Sweave.

Ostatnia linia informuje o tym, że przetwarzanie pliku `przyklad.Snw` przebiegło bez problemów. W katalogu roboczym utworzonych zostało kilka nowych plików między innymi plik `przyklad.tex`. To plik wynikowy, zapisany w języku LaTeX. W tym pliku został umieszczony opis oraz wyniki wykonania komend R znajdujących się w pliku `przyklad.Snw`. Plik `przyklad.tex` możemy skompilować używając kompilator języka LaTeX np. programem `pdflatex.exe` do formatu `pdf`. Wynik takiej kompilacji przedstawiony jest na rysunku 2.5.

Te fragmenty kodu (ang. *chunks*), z braku lepszego tłumaczenia, będziemy nazywać wstawkami.

Przyjrzyjmy się bliżej zawartości pliku `przyklad.Snw`. Wygląda on jak zwykły plik z kodem w LaTeXu, w którym dodatkowo występują fragmenty o składni:

```
<< argumenty >>=
kod R
@
```

Funkcja `Sweave::Sweave()` powoduje przetworzenie pliku, w tym przypadku `przyklad.Snw`. Wykonuje ona kod R znajdujący się we wstawkach, a wyniki wykonania tego kodu zapisuje do pliku wyjściowego (w tym przypadku do pliku `przyklad.tex`). Określając odpowiednie argumenty wstawek, możemy określać, w jaki sposób wyniki zostaną wklejone do pliku wynikowego. Wstawki można parametryzować. Jeżeli nie chcemy, by wyniki poleceń R były wpisywane do pliku wyjściowego, to za argumenty wstawki należy podać `echo=false,results=hide`.

Jeżeli wynikiem poleceń R jest wykres, to aby znalazł się on w pliku wynikowym, należy za argument wstawki podać `fig=true`. W jednej wstawce możemy wyświetlić dowolnie wiele wykresów. Wynik każdego z nich zostanie przetworzony do dwóch plików: jednego w formacie `pdf` i jednego w formacie `eps`. Do pliku wynikowego automatycznie zostanie dodana odpowiednia instrukcja LaTeX dołączająca plik z rysunkiem. W naszym przykładzie do pliku wynikowego zostanie dodana poniższa linia, włączająca wynik wykonania drugiego polecenia `boxplot()`.

```
\includegraphics{przyklad-002}
```

Jeżeli spodziewamy się, że wykonywana we wstawce funkcja R wyprodukuje kod w formacie LaTeX (a więc nie jest potrzebna dodatkowa transformacja wyniku), to w danej wstawce powinniśmy podać argument `results=tex`. Taki argument powinien być podany np. jeżeli korzystamy z funkcji `xtable::xtable()`, służącej do wyświetlania tabel w formacie LaTeX. Np. następująca wstawka

```
<< results=tex,echo=false >>=
library(xtable)
mat = matrix(1:25,5,5, dimnames=list(LETTERS[1:5], LETTERS[6:10]))
xtable(mat, caption="Tabela kolejnych liczb");
@
```

zostanie przetworzona na

```
% latex table generated in R 3.0.1 by xtable 1.5-2 package
% Thu Jul 10 11:13:41 2010
\begin{table}[ht]
\begin{center}
\begin{tabular}{rrrrrr}
    & F   & G   & H   & I   & J   \\  \hline
  \hline
  A &   1 &   6 &  11 &  16 &  21 \\
  B &   2 &   7 &  12 &  17 &  22 \\
  C &   3 &   8 &  13 &  18 &  23 \\
  D &   4 &   9 &  14 &  19 &  24 \\
  E &   5 &  10 &  15 &  20 &  25 \\   \hline
\end{tabular}
\caption{Tabela kolejnych liczb}
\end{center}
\end{table}
```

Do konwersji obiektów R do języka LaTeX można wykorzystać funkcję `toLatex()` lub funkcję `toBibtex()` przygotowującą odpowiednią tekstową reprezentacje w języku LaTeX dla obiektu będącego jej argumentem.

Jeżeli chcemy w kodzie LaTeXa wykonać wyrażenie napisane z użyciem składni programu R, to zamiast używać wstawek można posłużyć się poleceniem `\Sexpr`. Poniżej prezentujemy przykład kodu, w którym po przetworzeniu przez `Sweave()` prawa strona wyrażenia (fragment w poleceniu `\Sexpr{...}`) zostanie zastąpiona wynikiem wyliczonym w programie R. Poniższy skrypt

```
Wynik do wyliczenia w R należy umieścić w poleceniu \verb|\Sexpr|
$$   \frac{1-log(3)}{\sqrt 5} = \Sexpr{(1-log(3))/5^.5}   $$
```

w wyniku przetwarzania zostanie zamieniony na

```
Wynik do wyliczenia w R należy umieścić w poleceniu \verb|\Sexpr|
$$   \frac{1-log(3)}{\sqrt 5} = -0.0441007561757451   $$
```

Jeżeli nasz wynikowy plik ma być bardzo duży, to możemy nie chcieć, by wyniki wszystkich wstawek umieszczane były w jednym, tym samym pliku. Ustawienie we wstawce argumentu `split=TRUE` spowoduje, że wynik wykonanego kodu R zostanie zapisany w osobnym pliku o nazwie w postaci `plik-etykieta.tex`, gdzie etykieta powinna być określona w liście argumentów.

```
<<split=FALSE, label=komentarz.S>>
Ten tekst trafi do innego pliku, ktory zostanie wlaczony do glownego
    pliku LaTeX.
@
```

Wynik z wykonania skryptów R umieszczany jest w pliku wynikowym w jednym z dwóch środowisk `Sinput` (dla instrukcji R) i `Soutput` (dla wyników tych instrukcji). Aby LaTeX potrafił rozpoznać te środowiska musi znać ich definicje. Znajdują się one w pliku `Sweave.sty`, który jest w katalogu pakietu `Sweave`. Jeżeli w pliku wejściowym `.Snw` nie ma dołączenia tego pliku, to `Sweave()` automatycznie do preambuły dodaje linię

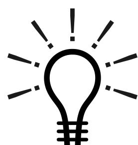
```
\usepackage{Sweave}
```

Jeżeli nie chcemy, by ta linia była dołączana, to możemy „oszukać” `Sweave`’a dodając do pliku `.Snw` zakomentowaną linię. Ta sztuczka przydaje się, jeżeli chcemy użyć innych np. własnych definicji wspomnianych środowisk.

```
% \usepackage{Sweave}
```

Korzystając z pakietu `Sweave` możemy wygenerować zarówno dokument pdf’a z objaśnieniem fragmentu kodu R (jak w przykładzie), automagicznie wygenerować raport z wyników naszej firmy (szef się ucieszy ;-), a my zaoszczędzimy czas),

jak i wygenerować prezentacje na konferencję np. korzystając z pakietu `beamer`. Raport możemy wygenerować do dokumentu Worda, Open Office, do formatu `HTML` itp. Więcej informacji znaleźć można w pomocy dla funkcji `R2HTML::HTML()`.



Programowanie objaśniające nadaje się świetnie do przedstawiania konkretnych rozwiązań/poleceń/funkcji w R. Opis rozwiązania połączony jest z kodem, który to rozwiązanie generuje, a również z wynikiem wykonania tego kodu. Czytelnik może więc szczegółowo porównać wynik z argumentami wywoływanych poleceń, mając pewność, że autor nie pominął jakiegoś ważnego parametru.
Takie przykładowe rozwiązania są dostępne w programie R w postaci ilustracji (ang. *vignette*). Do pakietów R, poza funkcjami, zbiorami danych i opisami, dołączane są również ilustracje rozwiązań określonych problemów. Ilustracja składa się z kodu R prezentującego rozwiązanie oraz pliku przedstawiającego wynik tego rozwiązania. Dostęp do ilustracji rozwiązań odbywa się z użyciem funkcji `vignette()`. Poniżej przedstawiamy przykładowe wywołania tej funkcji.

Wyświetlmy listę ilustracji dostępnych w zainstalowanych pakietach. Zobaczmy jedną konkretną ilustrację, dotyczącą wielowymiarowej oceny gęstości. Poniższe polecenie otworzy plik PDF z opisem funkcjonalności funkcji `ks::kde()` oraz opisem przykładów wykorzystania tej funkcji w programie R. Kolejne polecenie otworzy okno edytora z kodami w języku R, których wyniki zostały przedstawione w powyższej ilustracji.

```
vignette(all = TRUE)
library("ks")
ilustracja <- vignette("kde")
print(ilustracja)
edit(ilustracja, editor="internal")
```

## 2.7 Budowa aplikacji www z pakietem `shiny`

W poprzednim rozdziale pokazaliśmy, jak tworzyć automatyzowalne raporty, prezentacje, ilustracje kodu z użyciem pakietów `knitr` i `Sweave`. Jednym z częstych zastosowań raportów jest przygotowanie zbioru zestawień dla innej osoby (klienta, przełożonego, recenzenta, kolegi z zespołu). Wadą raportu jest jego statyczność i liniowość. Raz wygenerowany raport ma określoną treść, którą zazwyczaj przegląda się strona po stronie.

Co jednak, gdybyśmy chcieli w interaktywny sposób eksplorować dane? Dać odbiorcy możliwość wyboru interesujących go parametrów i wyznaczenia opisu dla zadanych parametrów? W takiej sytuacji często stosowanym rozwiązaniem jest zbudowanie aplikacji z różnymi przyciskami, pozwalającymi na interakcję z danymi. W programie R można taką interaktywną aplikację zbudować na wiele sposobów. Najłatwiejszym a jednocześnie sposobem o olbrzymich możliwościach jest skorzystanie z pakietu `shiny` [49].

Przedstawimy ten pakiet na przykładzie. Używając go stworzymy aplikację, którą będzie można otworzyć w przeglądarce internetowej. Dzięki temu możemy

samą aplikację umieścić na serwerze i udostępnić innym użytkownikom. Jeżeli z jakichś powodów nie chcemy jej udostępniać innym, możemy też z tą aplikacją pracować lokalnie, na własnym komputerze.

Model budowy aplikacji z użyciem `shiny` jest zgodny z modelem *akcja-reakcja* (ang. *reactive programming*). Jest to popularny model budowy aplikacji wokół pracy z danymi, znany np. z arkuszy kalkulacyjnych w których część komórek arkusza może zależeć od innych komórek. Zmieniając stan komórek wejściowych automatycznie zmienia się stan komórek zależnych (mówimy, że wartości są „odświeżone"). Kontrolując, które komórki zależą od których, możemy „odświeżyć" (czyli przeliczyć na nowo wartości) tylko tych komórek, które mogą się zmienić.

Podobnie będzie w aplikacji zbudowanej z biblioteką `shiny`, tyle że struktury zależności pomiędzy wejściem a wyjściem nie musimy jawnie opisywać, będzie ona odtworzona przez samą bibliotekę. Takie „inteligentne odświeżanie" tylko tych wartości, które mogą się zmienić pozwala na budowę szybkich i jednocześnie złożonych aplikacji. Wszystkie elementy aplikacji podzielone są na elementy wejściowe i wyjściowe. Model *akcja-reakcja* wymaga oprogramowania sposobu w jaki elementy wejściowe wpływają na wygląd/stan/wartość elementów wyjściowych. Jeżeli w trakcie pracy zmieniona zostanie wartość jakiegoś elementu wejściowego (kliknięty przycisk, wpisana wartość liczbowa, przesunięty suwak) to przeliczone zostaną odpowiednie elementy wyjściowe.

W określonych sytuacjach, w takim modelu wyróżnia się też tzw. przekaźniki. Jeżeli jakaś wartość wymaga dużej ilości czasu do obliczeń, a jednocześnie jest elementem kilku obiektów wyjściowych, to aby uniknąć wielokrotnego liczenia tej samej wartości stosuje się buforowanie w warstwie przekaźników, pomiędzy wejściem a wyjściem. W `shiny` służy do tego funkcja `reactive()`.

Zobaczmy jak to wygląda na przykładzie. Zaczniemy od instalacji i włączenia pakietu `shiny`.

```
install.packages('shiny')
library(shiny)
```

Aby zbudować w pełni funkcjonalną aplikację, wystarczy stworzyć dwa pliki, domyślnie o nazwach `ui.r` i `server.r`. W pliku `ui.r` określa się interfejs użytkownika, definiuje się elementy widoczne, określa się gdzie i jak będą wyglądać elementy wejściowe i elementy wyjściowe. W pliku `server.r` określa się funkcje-/transformacje, wyznaczające wartości elementów wyjściowych.

I tyle. Plik `ui.r` określa jak wygląda aplikacja, plik `server.r` określa jak aplikacja funkcjonuje.

Skoro to takie proste, to zbudujmy prostą aplikację, która odczyta z Internetu dane o cenach mieszkań w Warszawie a następnie przedstawi graficznie i tekstowo rozkład cen za metr kwadratowy. Aplikacja ma umożliwiać określenie dzielnicy Warszawy, którą chcemy analizować, oraz określenie powierzchni mieszkań, które nas interesują. Jako wynik chcemy zobaczyć jak wygląda rozkład cen za metr kwadratowy mieszkań z określonej dzielnicy o określonym rozmiarze.

W poniższym przykładzie wykorzystamy dane pochodzące z portalu `SmarterPoland.pl` [50]. Zacznijmy od ich wczytania.

```
mieszkaniaKWW2011 <-
     read.table("http://tofesi.mimuw.edu.pl/~cogito/smarterpoland/
          mieszkaniaKWW2011/mieszkaniaKWW2011.csv", row.names=NULL, sep=";
          ", header=TRUE, colClasses=c("factor", "factor", "numeric", "
          numeric", "factor", "numeric", "numeric", "factor", "Date"))
```

Zobaczmy co jest w dwóch pierwszych wierszach tego zbioru danych.

```
head(mieszkaniaKWW2011, 2)
##     miasto               ulica pokoi powierzchnia pietro   cena
## 1 Warszawa skwer wyszynskiego     2           56      3 550000
## 2 Warszawa              barska     2           50      2 497000
##   cenam2 dzielnica       data
## 1   9822      Wola 2011-09-13
## 2   9940    Ochota 2011-09-13
```

Wczytane dane zawierają informacje o cenach ofertowych różnych mieszkań z Warszawy i innych miast (te inne miasta nas na razie nie interesują) z lat 2007-2011. Analiza takich danych to bardzo ciekawe zadanie, ale wykraczające poza zakres tej książki. Skupimy się tutaj na wykorzystaniu tych danych do ilustracji działania biblioteki `shiny`.

Osoby zainteresowane analizami takich danych z użyciem modeli liniowych i mieszanych znajdą kilka ciekawych modeli w książce [51].

Zacznijmy od określenia interfejsu, a zarazem pliku `ui.r`. Chcemy oprogramować aplikację wyglądającą tak, jak na rysunkach 2.6 – 2.8. Posiadającą pole wyboru określające dzielnice i suwak do wyboru zakresu powierzchni, dwa panele na których będzie przedstawiony histogram i tekstowy opis cen metra kwadratowego.

Aplikację zawierającą blok wejścia i wyjścia, można utworzyć z użyciem funkcji `pageWithSidebar()`, której przekazuje się opis elementów wejściowych i wyjściowych. Zacznijmy od opisu elementów wyjściowych. W tym przykładzie do zdefiniowania pola wyboru użyjemy funkcji `selectInput()` (argumenty określają identyfikator tego wejścia, nazwę, listę możliwych wartości i wartość domyślną), a do zdefiniowania suwaka użyjemy funkcji `sliderInput()` (kolejne argumenty to identyfikator tego wejścia, zakres zmienności i wartości domyślne). Biblioteka `shiny` udostępnia wiele szablonów różnego typu elementów wejściowych (listy jednokrotnego lub wielokrotnego wyboru, pola tekstowe itp), patrz [49].

Elementy wyjściowe to dwa panele, odpowiadające dwóm zakładkom w aplikacji. Na pierwszym panelu wyświetlimy wykres z użyciem funkcji `plotOutput()` (argumentem jest identyfikator pola wyjściowego, o nazwie `wyjscieHistogram`). Na drugim panelu wyświetlimy sam tekst, używając funkcji `verbatimTextOutput()` (argumentem jest identyfikator wyjścia, tutaj `wyjscieTabela`). Biblioteka `shiny` ma dostępnych wiele szablonów różnego typu elementów wyjściowych, pozwalających na wyświetlenie tabeli, pobranie pliku, wyświetlenie kodu HTML itp.

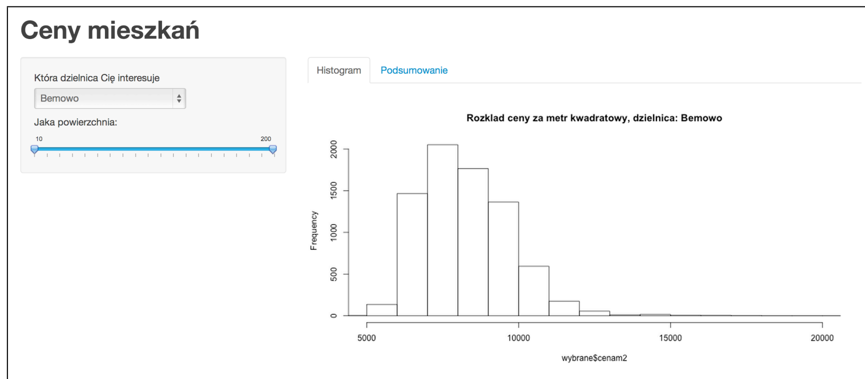
Nie ma sensu tej listy umieszczać tutaj, zbyt szybko się ona rozrasta. Pełną i aktualną listę można zawsze znaleźć w dokumentacji.

Poniżej znajduje się zawartość pliku `ui.r`, opisująca interfejs.

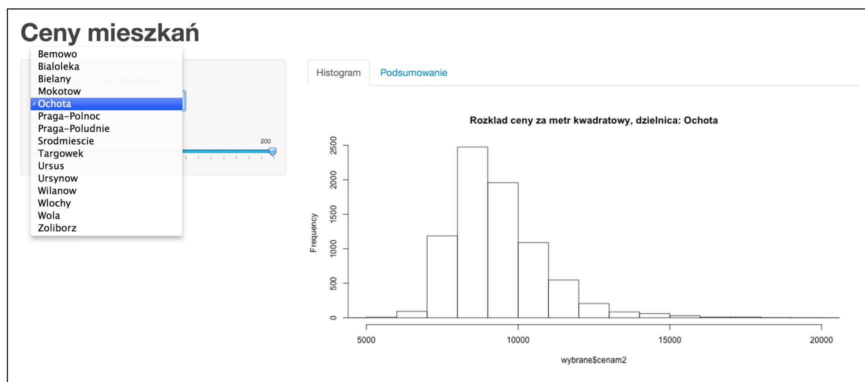
```
# plik: ui.r
shinyUI(pageWithSidebar(
  headerPanel("Ceny mieszkan"),

  sidebarPanel(
    selectInput(inputId = "dzielnica",
                label = "Ktora dzielnica Cie interesuje",
                choices = c("Bemowo", "Bialoleka", "Bielany", "Mokotow",
                    "Ochota", "Praga-Polnoc", "Praga-Poludnie", "
                    Srodmiescie", "Targowek", "Ursus", "Ursynow", "
                    Wilanow", "Wlochy", "Wola", "Zoliborz"),
                selected = "Bemowo"),
    sliderInput("zakres", "Jaka powierzchnia:",
                min = 10, max = 200, value = c(10,200), step= 10)),
```

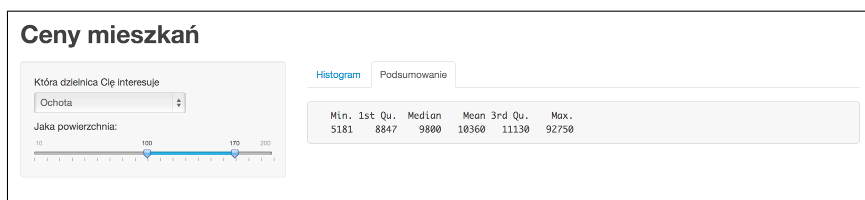
```
    mainPanel(
      tabsetPanel(
        tabPanel("Histogram", plotOutput("wyjscieHistogram")),
        tabPanel("Podsumowanie", verbatimTextOutput("wyjscieTabela"))
      ))
))
```




**Rysunek 2.6:** Początkowy wygląd oprogramowanej aplikacji. Po lewej stronie widoczne są dwa elementy wejściowe. Po prawej można przełączać się pomiędzy elementami wyjściowymi.




**Rysunek 2.7:** Rozkład cen dla mieszkań z Ochoty. Definicja z pliku `ui.r` została przekształcona na pole wyboru.




**Rysunek 2.8:** Zakładka z tekstowym wyjściem. Aby wyświetlić tę stronę uaktualniono wyjście o identyfikatorze `wyjscieTabela`.

W codziennej pracy lepiej dane wczytywać z lokalnego pliku. Przykład z wczytywaniem danych z Internetu ma tą zaletę, że po przepisaniu go będzie też działał na innych komputerach, bez konieczności instalowania dodatkowych plików.

Mając zdefiniowany interfejs, powinniśmy teraz zdefiniować transformacje. Opiszmy sposób wyznaczania wartości elementów wyjściowych, czyli zawartość pliku `server.r` (kod poniżej). Na początku tego pliku wczytujemy dane. Może to potrwać do kilkunastu sekund (dane pobierane są z Internetu). Pobieranie danych będzie wykonane tylko raz na sesję, uruchomiona aplikacja będzie pamiętała te dane w ramach swojej sesji. Następnie definiujemy funkcje generujące wartość elementów wyjściowych. Poniżej opisane są dwa elementy `wyjscieHistogram` i `wyjscieTabela`. Oba są tworzone na podstawie szablonów generujących wykresy (funkcja `renderPlot()`) i opisy tekstowe (funkcja `renderPrint()`).

Na poniższych przykładach w przypadku obu funkcji, najpierw z całego zbioru danych wybierane są tylko wiersze odpowiadające mieszkaniom z określonej dzielnicy i o określonej powierzchni. Następnie te wiersze są przedstawiane graficznie lub tekstowo. Warto zauważyć, że w opisie elementów wyjściowych wykorzystywane są wartości określone przez elementy wejściowe. Na poniższym przykładzie zmienna `input$dzielnica` przyjmuje wartość wybraną w polu wyboru. Wektor `input$zakres` to dwie liczby określone przez suwak aplikacji.

Cały kod przykładowego pliku `server.r` wygląda następująco.

```
# plik: server.r
mieszkaniaKWW2011 <-
  read.table("http://tofesi.mimuw.edu.pl/~cogito/smarterpoland/
      mieszkaniaKWW2011/mieszkaniaKWW2011.csv", row.names=NULL, sep=";",
      header=TRUE, colClasses=c("factor", "factor", "numeric", "numeric",
       "factor", "numeric", "numeric", "factor", "Date"))

shinyServer(function(input, output) {
  output$wyjscieHistogram <- renderPlot({
    wybrane <- mieszkaniaKWW2011[
                mieszkaniaKWW2011$dzielnica == input$dzielnica &
                mieszkaniaKWW2011$powierzchnia >= input$zakres[1] &
                mieszkaniaKWW2011$powierzchnia <= input$zakres[2], ]

    hist(wybrane$cenam2, xlim=c(5000,20000), main=paste("Rozklad ceny za
        metr kwadratowy", input$dzielnica))
  })

  output$wyjscieTabela <- renderPrint({
    wybrane <- mieszkaniaKWW2011[
                mieszkaniaKWW2011$dzielnica == input$dzielnica &
                mieszkaniaKWW2011$powierzchnia >= input$zakres[1] &
                mieszkaniaKWW2011$powierzchnia <= input$zakres[2], ]

    summary(wybrane$cenam2)
  })
})
```

Powyżej opisane dwa pliki to już cała aplikacja, której działanie przedstawione jest na rysunkach 2.6 – 2.8. Aby ją uruchomić, wystarczy użyć funkcji `runApp()` jako argument wskazując katalog z powyżej opisanymi plikami.

```
runApp('katalog_z_plikami_ui.r_server.r')
##
## Listening on port 8100
```

Jak widzimy, tworzenie własnych aplikacji jest bardzo proste, a praca do wykonania sprowadza się do zdefiniowania interfejsu oraz oprogramowania funkcji wyznaczających wartości wyjściowe. Dzięki tej prostocie biblioteka `shiny` w bardzo szybkim tempie zdobywa popularność.

## 2.8 Budowanie własnych pakietów

W programie R pakiety służą do grupowania zbiorów danych lub funkcji. Jeżeli opracowaliśmy kilka użytecznych funkcji lub zbiorów danych i chcemy je udostępnić innym osobom, to najlepszym sposobem będzie złożenie ich w pakiet. Taki pakiet możemy wysłać naszym studentom, współpracownikom, klientom, udostępniając im opracowane funkcje. Coraz popularniejsze staje się też dołączanie pakietów do publikacji proponujących nową metodę analizy danych. Dołączając pakiet umożliwiamy łatwy dostęp do opracowanej przez nas metody. Stworzony pakiet możemy umieścić na naszej stronie www, wysłać zainteresowanym osobom mailem lub, jeżeli chcemy by był dostępny szerokiemu gronu, możemy go umieścić w publicznym repozytorium pakietów R, np. repozytorium CRAN lub Bioconductor. Nawet jeżeli opracowaliśmy zbiór funkcji tylko do naszego użytku wciąż wygodnie jest złożyć w pakiet. Korzystanie z pakietów pozwala na łatwe przenoszenie kodu, automatyczne testowanie opracowanych funkcji, łatwe zarządzanie dokumentacją kodu, przenaszalność funkcjonalności pomiędzy różnymi systemami operacyjnymi, komputerami lub wersjami programu R.

Podsumowując: warto tworzyć pakiety!

Aby zbudować własny pakiet musimy wykonać kilka czynności:

- zainstalować dodatkowe oprogramowanie niezbędne do budowy pakietów,
- przygotować odpowiednią strukturę plików i katalogów,
- przetestować i zbudować opracowany pakiet.

Poniżej omówimy każdy z tych kroków. Szczegółowo proces budowy pakietów jest opisany w dokumencie [42]. Jest on obszerniejszy niż informacje z tego rozdziału, zawiera np. opis jak dołączyć do pakietu kompilowane źródła w innych językach, jak dołączyć dodatkową dokumentację, itp. W tym rozdziale omówimy proces tworzenia prostego przykładowego pakietu.

### 2.8.1 Niezbędne oprogramowanie

Skrypty budujące pakiety R wymagają dodatkowych programów, które nie są zawarte w podstawowej instalacji programu R. Możemy każdy z tych dodatkowych programów zainstalować samodzielnie lub wykorzystać gotowe zestawy zawierające niezbędne oprogramowanie w odpowiedniej wersji. Poniżej opiszemy, gdzie znaleźć taki zestaw. Dalsze postępowanie zależy od tego jakiego systemu operacyjnego używamy.

#### 2.8.1.1 Unix/Linux

W większości dystrybucji niezbędne programy są już zainstalowane.

#### 2.8.1.2 Windows

Zestaw dodatkowego oprogramowania dla systemu Windows można pobrać ze strony `http://www.murdoch-sutherland.com/Rtools/`. Ten zestaw jest nazywany „Rtools" i jest przygotowywany przez Duncana Murdocha dla każdej wersji R począwszy od 1.9 (wcześniej zajmował się tym Brian Ripley). Dla wersji 3.0 ten zestaw zawiera MinGW, Perl i kilka innych dodatków. Po zainstalowaniu wersji „Rtools" odpowiedniej do wersji R na którą chcemy budować pakiety, musimy wykonać jeszcze trzy kroki opisane w
`http://www.murdoch-sutherland.com/Rtools/Rtools.txt`.

- Zainstalować dystrybucję $\LaTeX$a. Jest ona niezbędna, by budować dokumentację w formacie pdf. Jedną z popularniejszych dystrybucji dla systemu Windows jest MikTeX, zobacz `http://www.miktex.org`.
- Zainstalować wsparcie do generowania plików pomocy w formacie HTML dla systemu Windows. Odpowiedni program można znaleźć na stronie `http://msdn.microsoft.com/en-us/library/ms669985.aspx`. Jest on niezbędny do generowania dokumentacji w formacie `.chm`.
- Zainstalować instalator Inno (`http://www.innosetup.com`). Ten krok potrzebny jest tylko jeżeli chcemy też przebudowywać cały program R. Do budowania pakietów instalator Inno nie jest potrzebny.

Po zainstalowaniu tych narzędzi, należy dodać odpowiednie ścieżki do zmiennych systemowych. Przyjmując, że `Rtools` zainstalowaliśmy do katalogu `c:\Rtools`, możemy to zrobić poleceniem:
`PATH=c:\Rtools\bin;c:\Rtools\perl\bin;c:\Rtools\MinGW\bin`.

#### 2.8.1.3 MacOS

Aby budować nowe pakiety w systemie MacOS X 10.4 (i wyżej) muszą być zainstalowane następujące programy:

- Xcode Developer Tools w wersji 3.1 lub nowszej,
- Kompilator GNU dla Fortrana,

oba można pobrać ze strony `http://r.research.att.com/tools/`.

W Internecie można znaleźć wiele wskazówek dotyczących instalacji i rozwijania programu R w systemach Windows i Linux. Dla użytkowników MacOSa takich przewodników jest mniej, ale wiele użytecznych informacji można znaleźć pod adresami `http://r.research.att.com/` oraz
`http://cran.r-project.org/bin/macosx/RMacOSX-FAQ.html`.

### 2.8.2 Przygotowanie pakietu

Pakiety dla programu R mogą zawierać funkcje napisane w języku R, funkcje napisane w innych językach, np. C++, fortran itp, dokumentacje w formacie `Rd`, `pdf` lub `tex`, grafikę, dane i inne komponenty. Aby można było z nich korzystać należy te komponenty odpowiednie opisać i umieścić je w odpowiednich katalogach.

Techniczny i szczegółowy opis możliwości i ograniczeń związanych z tworzeniem pakietów można znaleźć w dokumencie [42]. Poniżej przedstawimy przykład jak krok po kroku stworzyć, zbudować i zainstalować nowy pakiet. Na potrzebę przykładu stworzymy pakiet `PBImisc`, w którym umieścimy zbiór danych oraz funkcję rysującą histogram.

#### 2.8.2.1 Przygotowanie funkcji i zbiorów danych

W pierwszej wersji pakietu umieścimy jedną funkcję i jeden zbiór danych. Poniższy skrypt definiuje nową funkcję `hist2()`, która rysuje histogram i dorysowuje do niego jądrowy estymator gęstości. Ten skrypt definiuje również wektor o nazwie `petlen` zawierający 150 liczb losowych.
Zbior danych to 150 liczb losowych. Zdefiniujmy też przykładową funkcję.

```
petlen <- c(rnorm(100), rnorm(50,4))
hist2 <- function(x, breakes="Sturges", bw="nrd0", name="") {
 hist(x, probability=T, breaks=breakes, main=name)
 dtmp <- density(x, bw=bw)
 lines(dtmp$x, dtmp$y,lwd=3, col="red3")
 rug(x, col="red")
}
```

#### 2.8.2.2 Przygotowanie struktury katalogów

Dane w pakiecie rozmieszczone są w określonej strukturze katalogów. Zaawansowani użytkownicy niezbędną strukturę katalogów mogą stworzyć ręcznie. Tutaj jednak, na potrzeby tego przykładu, wykorzystamy do tego celu funkcję `package.skeleton()`. Pierwszym argumentem tej funkcji jest lista obiektów, które chcemy umieścić w pakiecie. Mogą to być funkcje lub zbiory danych lub dowolne inne obiekty R. Drugi argument to nazwa pakietu, który chcemy utworzyć.

W poniższym przykładzie utworzymy pakiet `PBImisc` zawierający funkcję `hist2` i wektor `petlen`.

```
package.skeleton(list=c("hist2", "petlen"), name="PBImisc")
## Creating directories ...
## Creating DESCRIPTION ...
## Creating Read-and-delete-me ...
## Saving functions and data ...
## Making help files ...
## Done.
## Further steps are described in './PBImisc/Read-and-delete-me'.
```

W efekcie działania funkcji `package.skeleton()`, w katalogu `R/R-2.3.0/bin/` został utworzony podkatalog `PBImisc` z automatycznie wygenerowaną minimalną strukturą podkatalogów. Ta minimalna struktura składa się z trzech podkatalogów:

- podkatalog `data` zawiera zbiory danych zapisane w binarnym formacie `rda`. Każdy plik odpowiada jednemu zbiorowi danych. Po zainstalowaniu i włączeniu pakietu `PBImisc` dane te automatycznie będą dostępne w przestrzeni nazw pakietu.

  Pliki binarne zapisane w formacie `rda` można odczytywać funkcją `load()`. Do formatu `rda` można zapisywać funkcją `save()`. Więcej informacji o tych funkcjach jest w rozdziale 1.6.4.

- podkatalog `R` zawiera definicje funkcji zapisane w postaci tekstowej. Podczas budowania pakietu wszystkie pliki z rozszerzeniem `.R` z podkatalog `R` będą wykonane inicjując przestrzeń nazw. W każdym pliku może być dowolna liczba definicji funkcji, jednak dla czytelności zalecane jest by jeden plik odpowiadał jednej funkcji.

- podkatalog `man` zawiera opisy zbiorów danych i funkcji zapisane w formacie `Rd`. Format `Rd` to tekstowy format opisu obiektu. W trakcie budowania pakietu będzie on automatycznie przekształcony do formatu HTML, TEX oraz `pdf`.

Wszystkie obiekty, które udostępniamy w pakiecie, powinny mieć przygotowane pliki pomocy. Plik `Rd` składa się z kilku sekcji. Nazwy sekcji różnią się w zależności od tego, czy opisywany jest zbiór danych czy funkcja.

**Plik pomocy dla zbioru danych**

Podstawowe sekcje dla plików pomocy opisujących zbiory danych to:

- sekcja `\name` określa nazwę danego pliku pomocy,
- każda sekcja `\alias` wskazuje alias tego pliku pomocy, wskazany plik pomocy zostanie otwarty, również po wpisaniu komendy `?_alias_`. Można umieścić więcej niż jedną sekcję alias, dlatego mówimy tutaj o opisie pliku pomocy, a nie opisie zbioru danych. Jeden plik pomocy może opisywać kilka zbiorów danych lub kilka tematów pomocy,
- sekcja `\title` zawiera tytuł opisu, najczęściej jest krótka i zawiera jedno lub dwa zdania,
- sekcja `\usage` pokazuje, jak wczytać dany zbiór danych,
- sekcja `\format` przedstawia strukturę zbioru danych,
- sekcja `\details` przedstawia szczegółowy opis zbioru danych, może być dowolnie długa,
- sekcja `\source` zawiera listę publikacji lub innych referencji, w których można znaleźć więcej informacji o zbiorze danych,
- sekcja `\examples` prezentuje przykłady użycia zbioru danych.

W naszym przykładzie plik `PBImisc/man/petlen.Rd` został wygenerowany automatycznie. Musimy samodzielnie wypełnić odpowiednie pola. Poniżej znajduje się przykładowa zawartość uzupełnionego pliku pomocy.

```
\name{petlen}
\alias{petlen}
\docType{data}
\title{
  Przykladowy zbior danych
}
\description{
  Przykladowy zbior danych zawierajacy 150 liczb losowych z dwumodalnego
      rozkladu
}
\usage{data(petlen)}
\format{
  Wektor 150 liczb
  num [1:150] 1.46 1.29 1.31 1.56 1.36 ...
}
\details{
  Wektor zostal wygenerowany poleceniem
  petlen <- c(rnorm(100), rnorm(50,4))
}
\source{
  Brak zrodel.
}
\examples{
  data(petlen)
  hist2(petlen)
}
\keyword{petlen}
```

**Plik pomocy dla funkcji**

Podstawowe sekcje dla plików pomocy opisujących funkcje to:

- sekcja `\name` opisuje nazwę danego pliku pomocy,
- każda sekcja `\alias` (można umieścić więcej niż jedną) wskazuje alias tego pliku pomocy, wskazany plik pomocy zostanie otwarty, również po wpisaniu komendy `?_alias_`,
- sekcja `\title` zawiera tytuł opisu, najczęściej jest to jedno lub dwa zdania,
- sekcja `\description` zawiera rozszerzony opis działania funkcji,
- sekcja `\usage` przedstawia sygnaturę funkcji wraz z listą argumentów i ich wartościami domyślnymi,
- sekcja `\arguments` zawiera opisy dla każdego z argumentów,
- sekcja `\details` przedstawia szczegółowy opis działania funkcji, może być dowolnie długa,

- sekcja `\value` opisuje jakiego typu obiekt jest zwracany przez funkcję, oraz jakie pola zawiera ten obiekt,
- sekcja `\seealso` zawiera odwołania do innych funkcji o podobnym działaniu,
- sekcja `\examples` prezentuje przykłady użycia danej funkcji.

W naszym przykładzie plik `PBImisc/man/hist2.Rd` został wygenerowany automatycznie, należy go uzupełnić. Poniżej znajduje się przykładowa treść tego pliku po uzupełnieniu.

```
\name{hist2}
\alias{hist2}
\title{
  Rozszerzony histogram
}
\description{
Funkcja hist2() rysuje histogram i dorysowuje do niego jądrowy estymator
gęstości, razem ze znacznikami odpowiadającymi kolejnym obserwacjom.
}
\usage{
  hist2(x, breakes = "Sturges", bw = "nrd0", name = "")
}
\arguments{
  \item{x}{wektor liczb}
  \item{breakes}{liczba przedziałów, przekazywany dalej do hist()}
  \item{bw}{szerokość okna, przekazywany dalej do density()}
  \item{name}{tytuł wykresu}
}
\details{
  Funkcja hist2() wywołuje kolejno funkcje: hist() do narysowania
histogramu, density() do wyznaczenia jądrowego estymatora gęstości
i rug() do dorysowania znaczników obserwacji.
}
\value{Nie zwraca wyniku}
\seealso{
Funkcja rysująca histogram \code{\link{hist}}
}
\examples{
  data(petlen)
  hist2(petlen)
}
\keyword{histogram}
```

Dodatkowo w podkatalogu `man` można umieścić plik pomocy dotyczący całego pakietu. W naszym przykładzie jest to plik `PBImisc-package.Rd`.

W katalogu opisującym pakiet muszą się znaleźć pliki tekstowe o nazwach `DESCRIPTION` i `NAMESPACE`. Plik `DESCRIPTION` utworzony będzie automatycznie przez funkcję `package.skeleton()`, plik `NAMESPACE` będziemy musieli dodać ręcznie. Poniżej krótko omówimy zawartość obu plików.

### Opis pakietu, plik DESCRIPTION

Plik `DESCRIPTION` zawiera podstawowe informacje o pakiecie, jego wersji, autorze pakietu, zależnościach od innych pakietów. Jeżeli zdecydujemy się umieścić pakiet na serwerze CRAN, to informacje z pliku `DESCRIPTION` będą umieszczone na stronie www z której będzie można pobrać pakiet.

Przykładowa zawartość pliku `DESCRIPTION`.

```
Package: PBImisc
Type: Package
Title: Zbior pomocniczych funkcji uzytkownika PBI
Version: 1.0
Date: 2013-07-01
Author: Przemyslaw Biecek
Maintainer: PBI <przemyslaw.biecek@gmail.com>
Description: Prywatny pakiet z pomocniczymi funkcjami
License: GPL 2
LazyLoad: yes
```

### Opis zawartości pakietu, plik NAMESPACE

Plik `NAMESPACE` zawiera informacje o tym, które funkcje mają zostać udostępnione dla użytkownika po włączeniu pakietu. Funkcje wymienione w tym pliku będą widoczne w głównej przestrzeni nazw. Pozostałe funkcje zdefiniowane w plikach z podkatalogu `R` będą funkcjami prywatnymi, dostępnymi jedynie w przestrzeni nazw pakietu. Z funkcji prywatnych korzystać mogą inne funkcje z tego samego pakietu.

Poniższa zawartość pliku `NAMESPACE` wskazuje, że jedyną udostępnianą funkcją ma być funkcja `hist2()`.

```
export(
     hist2
)
```

Jeżeli nie chcemy wymieniać z nazwy wszystkich funkcji to możemy użyć wyrażeń regularnych do wskazywania, które funkcje mają być udostępnione. Np. poniższy wpis w pliku `NAMESPACE` uczyni publicznymi wszystkie obiekty zaczynające swoje nazwy od przedrostka `hist`.

```
exportPattern("^hist*")
```

Jeżeli funkcja nie jest deklarowana jako publiczna, czyli nie jest wymieniona w pliku `NAMESPACE`, to można się do niej odwołać wykorzystując operator :::.

Przykładowo `PBImisc:::hist2` odwoła się do funkcji `hist2` z pakietu `PBImisc`, bez względu na to, czy jest ona publiczna czy nie.

### 2.8.3 Weryfikacja, budowanie i instalacja pakietu

Po przygotowaniu pakietu następuje zazwyczaj sekwencja czynności, takich jak budowanie, testowanie i instalacja pakietu. Poniżej omówiony jest każdy z tych kroków.

#### Budowanie pakietu

Zakładamy, że wszystkie narzędzia opisane w rozdziale 2.8 zostały zainstalowane. Przyjmijmy też, że pakiet który chcemy zbudować nazywa się `PBImisc` i znajduje się w katalogu `R/bin`. Otwieramy konsolę i przechodzimy do katalogu `R/bin` w którym znajduje się plik `R.exe`. Jeżeli wywołamy polecenie `R.exe` bez dodatkowych argumentów, to uruchomiony zostanie interpreter R. Jeżeli jednak pierwszym parametrem polecenia `R` będzie `CMD`, to uruchomiony będzie jeden z predefiniowanych skryptów perlowych, znajdujących się w katalogu `R/bin`. Jednym z takich skryptów jest `build`, który służy do budowania pakietów.

W poniższym przykładzie uruchamiamy skrypt `build` oraz budujemy pakiet `PBImisc`.

```
C:\_Soft_\R\R-3.0.1\bin>R CMD build PBImisc

* checking for file 'PBImisc/DESCRIPTION' ... OK
* preparing 'PBImisc':
* checking DESCRIPTION meta-information ... OK
* removing junk files
* checking for LF line-endings in source and make files
* checking for empty or unneeded directories
* building 'PBImisc_1.0.tar.gz'
```

Wynikiem wykonania polecenia `R CMD build PBImisc` jest utworzenie pliku `PBImisc_1.0.tar.gz`, który zawiera źródła pakietu. Ten plik może teraz być użyty do instalacji pakietu.

#### Testowanie pakietu

Do testowania pakietu można wykorzystać skrypt `check`. Skrypt ten sprawdzi, czy podany pakiet można zainstalować, czy dokumentacja generuje się bez błędów, czy przykłady dla funkcji udało się wykonać bez błędów, czy nie ma kolizji oznaczeń z innymi pakietami itp.

Jest to bardzo użyteczna funkcjonalność. Jeżeli dla naszych funkcji przygotujemy zestawy przykładów, które mogą służyć jako testy, to skrypt `check` umożliwi automatyczne testowanie funkcji z naszego pakietu.

Wynik procesu testowania przykładowego pakietu `PBImisc` przedstawiony jest poniżej.

```
C:\_Soft_\R\R-3.0.1\bin>R CMD check PBImisc
* checking for working pdflatex ... OK
* using log directory 'C:/_Soft_/R/R-3.0.1/bin/PBImisc.Rcheck'
* using~R version 3.0.1 (2013-04-14)
* using session charset: CP1250
* checking for file 'PBImisc/DESCRIPTION' ... OK
* checking extension type ... Package
* this is package 'PBImisc' version '1.0'
* checking package name space information ... OK
* checking package dependencies ... OK
* checking if this is a source package ... OK
* checking for .dll and .exe files ... OK
* checking whether package 'PBImisc' can be installed ... OK
* checking package directory ... OK
* checking for portable file names ... OK
* checking DESCRIPTION meta-information ... OK
* checking top-level files ... OK
* checking index information ... OK
* checking package subdirectories ... OK
* checking~R files for non-ASCII characters ... OK
* checking~R files for syntax errors ... OK
* checking whether the package can be loaded ... OK
* checking whether the package can be loaded with
       stated dependencies ... OK
* checking whether the name space can be loaded with
        stated dependencies ... OK
* checking for unstated dependencies in~R code ... OK
* checking S3 generic/method consistency ... OK
* checking replacement functions ... OK
* checking foreign function calls ... OK
* checking~R code for possible problems ... OK
* checking Rd files ... OK
* checking Rd metadata ... OK
* checking Rd cross-references ... OK
* checking for missing documentation entries ... OK
* checking for code/documentation mismatches ... OK
* checking Rd \usage sections ... OK
* checking data for non-ASCII characters ... OK
* checking examples ... OK
* checking PDF version of manual ... OK
```

Efektem ubocznym jest utworzenie katalogu `PBImisc.Rcheck`, w którym znajduje się zainstalowany pakiet, dokumentacja wygenerowana w rożnych formatach oraz wyniki wykonywania przykładów. Jeżeli chcemy umieścić pakiet w publicznym repozytorium CRAN, proces testowania pakietu musi przechodzić przez jakichkolwiek błędów czy ostrzeżeń.

### Instalacja pakietu

Pakiet możemy zainstalować na dwa sposoby. Pierwszy dotyczy instalacji pakietu ze źródeł. Ten proces wymaga zainstalowanych programów o których pisaliśmy w rozdziale 2.8. Użytkownicy systemów Windows i MacOS mogą również instalować już zbudowane pakiety. W tym przypadku nie jest potrzebne dodatkowe oprogramowanie, musimy mieć jednak pakiet zbudowany dla tej wersji `R` z której korzystamy. Opiszemy obie możliwości.

- Instalacja pakietu ze źródeł

  Do instalacji pakietu można wykorzystać skrypt `INSTALL`. Skrypt ten zainstaluje pakiet ze źródeł dostępnych albo w katalogu albo w spakowanym pliku o rozszerzeniu `tar.gz` (zobacz rozdział o budowaniu pakietu). Wynik instalacji pakietu `PBImisc` ze źródeł dostępnych w spakowanym pliku `PBImisc.tar.gz` przedstawiony jest poniżej.

  ```
  C:\_Soft_\R\R-3.0.1\bin>R CMD INSTALL PBImisc_1.0.tar.gz
  * installing to library '/R/win-library/3.0.1'
  * installing *source* package 'PBImisc' ...
  **~R
  ** data
  ** preparing package for lazy loading
  ** help
  *** installing help indices
  ** building package indices ...
  ** MD5 sums
  * DONE (PBImisc)
  ```

  Podobny efekt można uzyskać poleceniem `R CMD INSTALL PBImisc`, która zainstaluje pakiet z katalogu `PBImisc`.

  Pakiet możemy też zainstalować z linii poleceń interpretera programu R wywołując funkcję `install.packages()`. Należy jako pierwszy argument podać ścieżkę do pliku `tar.gz`, a jako argument `repos` podać wartość `NULL`.

  ```
  install.packages("C:/_Soft_/R/PBImisc_1.0.tar.gz", repos=NULL)
  ```

W niektórych wersjach systemu Windows pakiety instalowane są do lokalnych katalogów, np. w polskojęzycznym systemie Windows Vista, jeżeli użytkownik nie ma praw zapisu do katalogu Program Files to dodatkowe pakiety instalowane są w ścieżce C:/Users/pbiecek/Documents/R/win-library/3.0/.

- Instalacja skompilowanego pakietu

  Przypuśćmy, że chcemy przygotować pakiet dla R i rozesłać go innym osobom by mogły go użyć. Wiemy, że te osoby używają systemu Windows lub MacOSa i nie chcemy zmuszać ich do instalacji dodatkowego oprogramowania. W takiej sytuacji najlepiej przygotować skompilowaną wersję pakietu.

  Aby to zrobić instalujemy dany pakiet lokalnie na naszym komputerze. Instalacja pakietu spowoduje, że w podkatalogu `R/library` zostanie dodany nowy podkatalog z nazwą pakietu, np. `PBImisc`. Wystarczy teraz ten katalog spakować do formatu `zip` do pliku `PBImisc.zip` i ten plik wysłać zainteresowanym osobom.

Tak przygotowany pakiet można zainstalować w linii poleceń interpretera R używając funkcji `install.packages()`. Należy jako pierwszy argument podać ścieżkę do pliku `zip` a jako argument `repos` podać wartość `NULL` (podobnie jak w poprzednim punkcie, tyle że plik ma rozszerzenie `.zip`).

Jeżeli chcemy umieścić nasz pakiet w repozytorium CRAN, to wystarczy skopiować zbudowany pakiet w postaci pliku `tar.gz` na serwer ftp o adresie `ftp://cran.r-project.org/incoming` (użytkownik anonymous). Oczywiście wcześniej należy upewnić się, że pakiet przechodzi testy bez żadnego błędu i bez żadnego ostrzeżenia. Jeżeli wysłany pakiet podczas sprawdzenia przez komendę `R CMD check` wygeneruje jakiekolwiek ostrzeżenie lub błąd, wtedy nie zostanie umieszczony w repozytorium CRAN.

Polityka firmy.

Powyżej przedstawiliśmy przykład budowania i instalowania prostego pakietu bez dokumentacji w postaci winietek, bez kodów w innych językach programowania i innych rozbudowanych funkcji. Więcej informacji o bardziej zaawansowanych możliwościach budowy pakietów można znaleźć np. w dokumentach:

- Opis procesu przygotowania pakietu `Writing R Extensions`,
  `http://cran.r-project.org/doc/manuals/R-exts.pdf`.
- Krótkie wprowadzenie `Creating R Packages: A Tutorial`,
  `http://epub.ub.uni-muenchen.de/6175/`.
- Inne, równie krótkie wprowadzenie `Creating R Packages, Using CRAN, R-Forge, And Local R Archive Networks And Subversion Repositories`,
  `http://cran.r-project.org/doc/contrib/Graves+DoraiRaj-RPackageDevelopment.pdf`.

## 2.9 Debugger i profiler

Jeżeli ktoś twierdzi, że pisze programy bez błędów, to znaczy, że nie pisał dużych programów.

Bez względu na doświadczenie, umiejętności, wiedzę i wiek, nie da się pisać dużych programów zupełnie pozbawionych błędów. Nawet program, który wydawał się być wolny od błędów, po zainstalowaniu nowej wersji jakiegoś zależnego pakietu może przestać działać lub zgłaszać bardzo dziwne wyniki. Należy więc nauczyć się wyszukiwać i naprawiać błędy. Teoretycznie można błędy wyszukiwać patrząc uważnie w kod i analizując go spokojnie linia po linii. Ale zazwyczaj znacznie łatwiej jest użyć debuggera.

### 2.9.1 Debugger

Narzędzia do wyszukiwania i poprawiania błędów nazywa się debuggerami. W programie R błędy można śledzić na wiele sposobów, np. używając funkcji z pakietu `debug`. Poniżej przedstawimy także kilka innych sposobów.

Sytuacje awaryjne w programie R raportowane są jako błędy lub jako ostrzeżenia. Wystąpienie błędu powoduje wywołanie funkcji `stop()`, co zazwyczaj kończy wykonywanie podprogramu. Przerywane jest wykonanie pętli, funkcji, bloku

As a general rule (subject to numerous exceptions, caveats, etc.):
1) it is programming and debugging time that most impacts 'overall' program execution time;
2) this is most strongly impacted by code readability and size (the smaller the better);
3) both of which are enhanced by modular construction and reuseability, which argues for avoiding inline code and using separate functions.
These days, I would argue that most of the time it is program clarity and correctness (they are related) that is the important issue, not execution speed.

Berton Gunter (in a discussion about parsing speed) fortune(125)

kodu i innych instrukcji. Ostrzeżenia (ang. *warnings*) też są raportowane, ale nie powodują przerwania wykonywania podprogramu. Po napotkaniu błędu i wywołaniu funkcji `stop()` wykonana zostanie akcja określona przez globalny parametr `error`. Zmieniając wartość tego parametru możemy zażądać, aby R przed przerwaniem wykonywanych instrukcji zapisał stan wszystkich aktualnych zmiennych. Umożliwi to analizę *post-mortem* tego, co się stało i ułatwi znalezienie przyczyny wystąpienia błędu.

Wartość globalnie zdefiniowanej opcji `error` określa, co ma się dziać po napotkaniu błędu. Tę wartość można zmieniać korzystając z argumentu `error` funkcji `options()`. Po poniższym poleceniu, program R po napotkaniu błędu zapisze stan do pliku `errorDump.RData`.

```
options(error=quote(dump.frames("errorDump",TRUE)))
```

Na potrzeby poniższych przykładów zdefiniujmy funkcję wyznaczającą logarytm.

```
funkcja <- function(x) {
   log(x)
}
```

Jeżeli wywołamy tę funkcję z argumentami złego typu, to zgłoszony zostanie błąd.

```
funkcja("napis")
## log: Using log base e.
## Error in log(x) : Non-numeric argument to mathematical function
## Execution halted
```

Po napotkaniu błędu (a więc przy próbie wyznaczenia logarytmu z napisu) zostało wywołane polecenie `dump.frames("errorDump", TRUE)`. Zapisało ono stan zmiennych w programie R do pliku `errorDump.rda`. Możemy teraz odczytać ten stan i przeanalizować go funkcją `debugger()`.

Poniżej przedstawiamy przykładową sesję z wbudowanym do programu R debuggerem. Zaczniemy od wczytania zawartość pliku `errorDump.rda`. Następnie uruchamiamy debugger na odczytanym środowisku.

```
load("errorDump.rda")
debugger(errorDump)
## Execution halted
## Available environments had calls:
## 1: funkcja("napis")
## 2: log(x)
## 3: .execute(.Primitive("log"), x, envir = sys.parent(1))
## Enter an environment number, or 0 to exit  Selection: 1
## Browsing in the environment with call:
##    funkcja("napis")
## Called from: debugger.look(ind)
```

Sprawdźmy wartość zmiennej `x` w tym miejscu

```
Browse[1]> x
## [1] "napis"
Browse[1]>
```

Pusta linia powraca do menu wyboru środowisk. Wybranie wartości `0` kończy pracę debuggera.

```
## Available environments had calls:
## 1: funkcja("napis")
## 2: log(x)
## 3: .execute(.Primitive("log"), x, envir = sys.parent(1))
##
## Enter an environment number, or 0 to exit  Selection: 0
```

Po uruchomieniu funkcji `debugger()` wyświetla się lista dostępnych przestrzeni nazw związanych z wywoływaniami kolejnych funkcji. W rozważanym przypadku, błąd został zgłoszony przez wbudowaną w programie R funkcję oznaczoną `.Primitive("log")`. Ta wbudowana funkcja została wywołana przez funkcję `log()`, a ta z kolei przez funkcję `funkcja()`. Pierwsze pytanie debuggera dotyczy numeru przestrzeni nazw, którą chcemy prześledzić. W powyższym przykładzie wybieramy `1`, a więc przestrzeń nazw widzianą przez funkcję `funkcja()`. Po tym wyborze linia konsoli poleceń rozpoczyna się prefixem `Browse[1]>`, który oznacza, że jesteśmy w przestrzeni nazw opatrzonej indeksem 1. Możemy teraz sprawdzić wartości zmiennych w tej przestrzeni nazw w chwili, w której pojawił się błąd. W naszym przykładzie sprawdziliśmy wartość zmiennej `x`, czyli argumentu funkcji `funkcja()`. Okazało się, że jest to napis `"napis"`, a więc wiemy już na czym polegał problem! Możemy teraz opuścić debugger klikając `ENTER`, a następnie wybierając przestrzeń nazw o numerze `0`.

Zaletą debugowania z zapisywaniem całego środowiska R jest to, że jeżeli błąd pojawił się podczas wykonywania programu w trybie wsadowym lub na innym komputerze, to możemy stan środowiska w chwili wystąpienia błędu skopiować i odtworzyć na inny komputerze, tam przeanalizować ten stan lub wysłać go komuś z prośbą o pomoc. Gdybyśmy chcieli, by po napotkaniu błędu natychmiast pojawiło się okienko debuggera, możemy skorzystać z funkcji `recover()`.

```
options(error = recover)
```

Teraz pojawienie się błędu automatycznie spowoduje otworzenie się okna debuggera.

Jeżeli błąd już wystąpił, a my nie zapisaliśmy informacji o stanie zmiennych w chwili wystąpienia błędu, to do zorientowania się, gdzie błąd mógł wystąpić, możemy użyć funkcji `traceback()`. Wypisuje ona listę ostatnio wywołanych funkcji wraz z wartościami argumentów w wywołaniu. Aby zobaczyć jak ta funkcja działa, zdefiniujmy jeszcze jedną pomocniczą funkcję, której wywołanie doprowadzi do błędu.

```
funkcja2 <- function(x,y) {funkcja(x); funkcja(y)}
funkcja2(1,"jeden")
## Error in log(x) : Non-numeric argument to mathematical function
```

Używając teraz funkcji `traceback()` możemy prześledzić, które funkcje i z jakimi argumentami były wywoływane w miejscu gdzie wystąpił błąd.

```
traceback()
## 2: funkcja(y)
## 1: funkcja2(1, "jeden")
```

Jeszcze innym sposobem szukania błędów jest debugowanie funkcji krok po kroku. Do tego celu służy funkcja `debug()`. Zaznacza ona, że wskazana argumen-

tem funkcja ma być uruchamiana w trybie śledzenia krok po kroku. W trakcie wykonywania tej funkcji można korzystać z poleceń: `n` (wykonaj kolejną linię kodu), `c` (wykonaj całą funkcję) lub `Q` (zamknij debugger). Debugowanie wyłącza się funkcją `undebug()`. Na poniższym przykładzie prezentujemy działanie funkcji `debug()`.

W pierwszej linii zaznaczamy funkcję `funkcja2()` do debugowania. Następnie wywołujemy tę funkcję, przez co uruchamia się też debugger.

```
debug(funkcja2)
funkcja2(1, "jeden")
## debugging in: funkcja2(1, "jeden")
## debug: {
##     funkcja(x)
##     funkcja(y)
## }
```

W tej chwili mamy dostęp do środowiska sprzed uruchomienia `funkcja2` (ale po zainicjowaniu argumentów). Sprawdźmy wartość `x`

```
Browse[1]> x
## [1] 1
```

Polecenie `n` wykonuje kolejną linię.

```
Browse[1]> n
## debug: funkcja(x)
```

A teraz sprawdźmy zawartość zmiennej `y`.

```
Browse[1]> y
## [1] "jeden"
```

Idziemy dalej dwa kroki i napotykamy na błąd.

```
Browse[1]> n
## debug: funkcja(y)
Browse[1]> n
## Error in log(x) : Non-numeric argument to mathematical function
```

Błędy i ostrzeżenia możemy zgłaszać używając funkcji `simpleError()` i `simpleWarning()`. Obie znajdują się w pakiecie `base` i obie służą do przygotowania komunikatu o błędzie lub ostrzeżeniu. W obu, za pierwszy argument należy podać komunikat błędu. Błąd lub ostrzeżenie można wygenerować funkcjami `stop()` lub `warning()`. Poniżej przedstawiamy przykłady użycia wymienionych funkcji.

Poniższa funkcja generuje błąd lub ostrzeżenie. Argument, który nie jest liczbą wywoła błąd.

```
drazliwa <- function(x) {
  if (class(x) != "numeric")
     stop(simpleError("miala byc liczba!!!"))
  if (x <0)
     warning(simpleWarning("spodziewalam sie czegos nieujemnego"))
}
drazliwa("test")
## Error: miala byc liczba!!!
```

Jeżeli argumentem będzie liczba ujemna, wygenerowane zostanie ostrzeżenie.

```
drazliwa(-2)
## Warning in drazliwa(-2): spodziewalam sie czegos nieujemnego
```

Jeżeli spodziewamy się ostrzeżeń w uruchamianym poleceniu i chcemy te ostrzeżenia zignorować, to możemy użyć funkcji `suppressWarnings()`. Argumentem tej funkcji jest polecenie do wykonania. Wyznaczony zostanie jego wynik, a dodatkowo zignorowane zostaną wszystkie wygenerowane ostrzeżenia.

Podobnie jest z błędami. Jeżeli spodziewamy się, że dane polecenie może wywołać błąd i na ten błąd chcemy odpowiednio zareagować (inaczej niż przerwaniem podprogramu), to możemy użyć funkcji `try()`. Ta funkcja wykona polecenie, podane jako jej pierwszy argument. Dodatkowo, jeżeli polecenie wykona się bez błędu, to wynikiem funkcji `try()` będzie wyznaczony wynik tego polecenia. Jeżeli podczas wykonywania wystąpi błąd, to wynikiem funkcji `try()` będzie obiekt klasy `try-error` zawierający komunikat o błędzie. Jeżeli nie chcemy, by komunikat o błędach był sygnalizowany, to należy za drugi argument funkcji `try()` podać `silent=TRUE`. Poniżej przykład bezpiecznego uruchomienia funkcji `cor()`.

Nawet, jeżeli funkcja `cor()` wygeneruje błąd, to błąd ten zostanie wychwycony przez funkcję `try()` i nie przerwie działania programu.
Przykład bezpiecznej korelacji, czyli takiej, która nie generuje błędów, a tylko wypisuje komentarz. Wywołanie zwykłej funkcji `cor()` zakończy się błędem. Warunek `class(tmp)=="try-error"` to sprawdzenie czy został zgłoszony błąd.

```
bezpieczna.cor <- function(x) {
   tmp <- try(cor(x), silent=TRUE)
   if (class(tmp)=="try-error") {
     cat("Korelacja nie zostala policzona\n")
     tmp <- 0
    }
   tmp
}
```

Przykładowe uruchomienie.

```
cor(1)
## Error in cor(1) : supply both 'x' and 'y' or a matrix-like 'x'
```

Wywołanie bezpiecznej wersji funkcji `cor()` nie sygnalizuje błędu.

```
bezpieczna.cor(1)
## Korelacja nie zostala policzona
## [1] 0
```

Przy okazji omawiania sytuacji błędogennych warto wspomnieć o pewnych, nieintuicyjnych, zagrożeniach wynikających z arytmetyki zmiennoprzecinkowej. Przyjrzyjmy się poniższemu przykładowi.

```
(v <- seq(0.7, 0.8, by=0.1))
## [1] 0.7 0.8
```

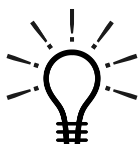
Wynik poniższej instrukcji powinien być zaskoczeniem.

```
v == 0.8
## [1] FALSE FALSE
```

Ponieważ różnica pomiędzy `0.8` a `0.7+0.1` jest większa od `double.neg.eps`, dlatego program R traktuje je jako różne liczby. Obie wartości `double.neg.eps` i `double.eps` określają maszynowy epsilon, czyli maksymalną różnicę pomiędzy liczbami, do której dwie liczby uznawana są za równe.

```
(0.7+0.1) - 0.8
## [1] -1.110223e-16
```

Powyższa różnica jest większa równa niż zmienna `double.neg.eps`. Jeszcze większe błędy numeryczne są widoczne przy potęgowaniu, np. ciekawemu czytelnikowi polecam sprawdzenie ile wynosi $\left(\sqrt{2}\right)^2 - 2$.



Należy być ostrożnym porównując liczby lub wektory liczb rzeczywistych. Problemy, wynikające z zaokrągleń oraz dokładności reprezentacji liczb rzeczywistych w pamięci komputera, mogą pojawić się nawet w bardzo prostych i niewinnie wyglądających sytuacjach. W przykładzie z poprzedniej strony spodziewaliśmy się wyniku `FALSE TRUE`. Zaskakujący wynik `FALSE FALSE` jest efektem zaokrągleń w arytmetyce liczb rzeczywistych. Zarówno 0.7 jak i 0.1 nie ma dokładnej skończonej reprezentacji w postaci binarnej, są więc zapisane w sposób przybliżony. Dodając te dwie niedokładnie reprezentowane liczby błędy się kumulują i dlatego ich suma jest reprezentowana jako liczba różna od 0.8. Aby uniknąć tego typu błędów najlepiej nie używać operatora == dla liczb rzeczywistych.
Jeżeli chcemy mieć pewność, że super-dokładnie wyznaczymy wynik, to należy korzystać z obliczeń symbolicznych lub obliczeń na liczbach całkowitych (np. w banku licząc pieniądze nie możemy pozwolić sobie na żadne niedokładności).

Warto przy okazji wspomnieć o dwóch zmiennych: `.Machine` i `.Platform`, które przechowują informacje o parametrach systemu operacyjnego, w którym pracujemy oraz o parametrach maszyny obliczeniowej, na której pracujemy. Każda z tych zmiennych jest listą wartości opisujących właściwości maszyny lub platformy, a więc jak reprezentowane są liczby w pamięci maszyny oraz z jaką dokładnością wykonywane są operacje. Dokładny opis poszczególnych pól tych list znaleźć można w pomocy dla obiektu `.Machine`.

```
unlist(.Machine)
##           double.eps        double.neg.eps             double.xmin
##         2.220446e-16          1.110223e-16           2.225074e-308
##           double.xmax           double.base           double.digits
##         1.797693e+308          2.000000e+00             5.300000e+01
##           double.rounding       double.guard       double.ulp.digits
##         5.000000e+00          0.000000e+00            -5.200000e+01
##         double.neg.ulp.digits dube.exponent           double.min.exp
##         -5.300000e+01          1.100000e+01            -1.022000e+03
##         double.max.exp           integer.max              sizeof.long
##         1.024000e+03          2.147484e+09             4.000000e+00
##          sizeof.longlong     sizeof.longdouble       sizeof.pointer
##         8.000000e+00          1.200000e+01             4.000000e+00
```

I zmienna `.Platform`.

```
unlist(.Platform)
##       OS.type     file.sep   dynlib.ext          GUI       endian
##     "windows"          "/"       ".dll"       "Rgui"     "little"
##       pkgType     path.sep       r_arch
##   "win.binary"         ";"           ""
##
```

### 2.9.2 Profiler

Profiler to narzędzie pozwalające analizować zajętość czasu procesora oraz użycie pamięci przy wykonaniu funkcji programu R. Narzędzie to może być bardzo przydatne do sprawdzenia, które elementy programu są czaso- lub pamięciożerne. Wyniki profilera można wykorzystać, aby przepisać wybrany problematyczny fragment kodu w inny sposób, tak by poprawić wydajność całego programu. W przypadku bardzo czasochłonnych obliczeń może być opłacalne napisanie wrażliwego fragmentu w języku kompilowanym, np. w C i wywoływanie takich zewnętrznych funkcji z programu R.

RAM is cheap and thinking hurts.

Uwe Ligges (about memory requirements in R)
fortune(192)

Profiler uruchamia się funkcją `Rprof()`. Pierwsze wywołanie tej funkcji powoduje włączenie zapisywania informacji o działaniu podprogramu do pliku określonego przez argument `filename` (domyślnie do pliku `"Rprof.out"`). Aby przerwać zapisywanie informacji dla profilera, należy uruchomić funkcję `Rprof` z argumentem `NULL`. Wyniki profilera można odczytać funkcją `summaryRprof()`. Poniżej przedstawiamy przykład wykorzystania profilera.

W pierwszym kroku definiujemy funkcję generującą po 1 000 000 liczb z różnych rozkładów i funkcję wypisującą te liczby na ekranie.

```
generuj <- function() {
  runif(10^6); rexp(10^6); rnorm(10^6); 1
}
wypisuj <- function() {
  cat(1:10^6)
}
```

Następnie włączamy profiler i wywołujemy dziesięć razy obie powyższe funkcje. Dane diagnostyczne z profilera są zapisywane do pliku `profiler.out`.

```
Rprof("profiler.out", interval = 0.01, memory.profiling=TRUE)
for (i in 1:10) {
     generuj()
     wypisuj()
}
```

Na koniec wyłączamy profiler.

```
Rprof(NULL)
```

Użyjmy teraz funkcji `summaryRprof()` do analizy zebranych wyników. Okazuje się, że najwolniejsza jest funkcja wypisująca na ekranie. Również relatywnie dużo czasu procesora zajmuje generowanie liczb z rozkładu normalnego, najszybciej generują się liczby z rozkładu jednostajnego.

```
summaryRprof("profiler.out", memory="both")
## $by.self
##         self.time self.pct total.time total.pct mem.total
## cat         15.72     85.3      15.73      85.4       4.8
## rnorm        1.36      7.4       1.36       7.4       9.5
## rexp         0.79      4.3       0.79       4.3       9.5
## runif        0.53      2.9       0.53       2.9       9.5
## :            0.01      0.1       0.01       0.1       0.5
## Rprof        0.01      0.1       0.01       0.1       0.0
## wypisuj      0.00      0.0      15.73      85.4       4.8
## generuj      0.00      0.0       2.68      14.5      28.6
##
## $by.total
##         total.time total.pct mem.total self.time self.pct
## cat          15.73      85.4       4.8     15.72     85.3
## wypisuj      15.73      85.4       4.8      0.00      0.0
## generuj       2.68      14.5      28.6      0.00      0.0
## rnorm         1.36       7.4       9.5      1.36      7.4
## rexp          0.79       4.3       9.5      0.79      4.3
## runif         0.53       2.9       9.5      0.53      2.9
## :             0.01       0.1       0.5      0.01      0.1
## Rprof         0.01       0.1       0.0      0.01      0.1
##
## $sampling.time
## [1] 18.42
```

Plik profiler.out jest niepotrzebny i można go usunąć.

```
unlink("profiler.out")
```

Na powyższym przykładzie porównywaliśmy czasy wykonania dla trzech funkcji, generujących liczby losowe. Wyniki profilera zostały zapisane we wskazanym pliku (tu `"profiler.out"`). Na załączonym przykładzie najwięcej czasu na wykonanie potrzebowała funkcja `cat()` (około 85% całkowitego czasu wywołania), funkcje wejścia/wyjścia z reguły są kosztowne czasowo. Tak jest i w tym przykładzie. Z generatorów funkcji losowych najwięcej czasu potrzebował generator liczb z rozkładu normalnych, a najkrócej generator dla rozkładu jednostajnego, co jest oczekiwanym wynikiem. Najwięcej pamięci potrzebowała funkcja `generuj()` (średnio, prawie 29MB na każde wykonanie).

Łatwiejszym sposobem sprawdzenia, jak długo wykonuje się pewien fragment kodu, jest użycie funkcji `system.time()` lub `proc.time()`. Te funkcje nie dają tak szczegółowych informacji jak profiler, ale jest łatwiej ich użyć. Wynikiem funkcji `proc.time()` jest wektor trzech liczb, określający ile czasu procesora zużyło środowisko R od początku pracy do danej chwili, ile czasu procesora zużył system oraz ile czasu minęło od uruchomienia środowiska R. Do wyznaczania ile czasu trwało wykonanie określonego polecenia funkcja `system.time()` korzysta z dwóch wywołań funkcji `proc.time()`, przed i po zakończeniu działania interesującego polecenia. Poniższy przykład pokazuje, jak używać funkcji `system.time()`. Przedstawione są wyniki porównania czasów czterech sposobów wypełnienia wektora liczbami losowymi. Przy okazji zauważmy, że najszybsze są operacje na całych wektorach.

Oznacza to, że jeżeli już trzeba używać pętli, to wewnątrz pętli wektory nie powinny być dynamicznie powiększane.

Zaczynamy od przykładu z dynamicznym powiększaniem wektora. Takie rozszerzanie wektora przez doklejanie zajmuje ponad 62 sekundy .

```
system.time({ x=NULL; for (i in 1:10^5) x =c(x, runif(1)) })
##    user  system elapsed
##   62.54    0.16   62.94
```

Ta wersja jest tak samo czasożerna, choć zapis jest nieznacznie krótszy.

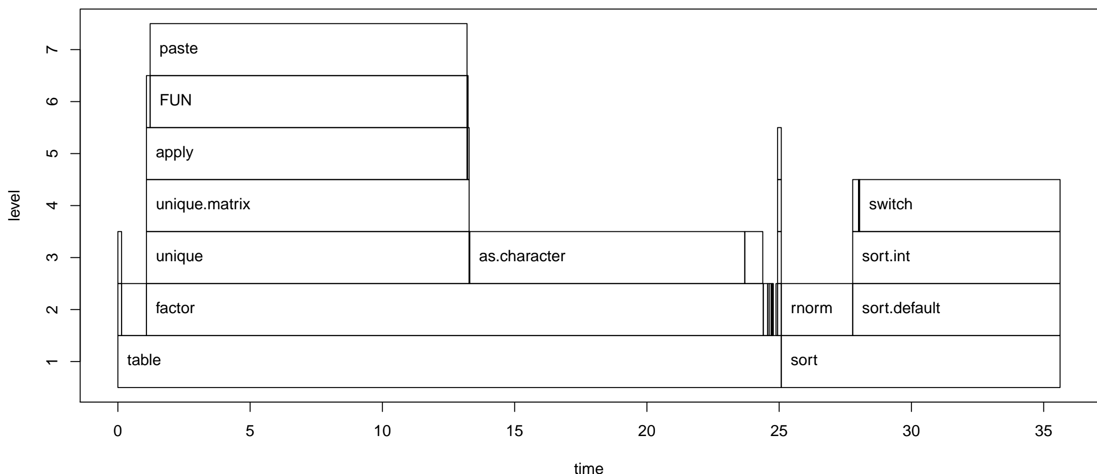
```
system.time({ x=NULL; for (i in 1:10^5) x[i] =runif(1) })
##    user  system elapsed
##   64.04    0.20   65.75
```

Jeżeli przed pętlą zainicjujemy wektor `x` o rozmiarze $10^5$ elementów, to ponad 50 razy skrócimy czas wykonywania się tego fragmentu kodu. Tylko dzięki uniknięciu dynamicznego rozszerzania wektora.

```
system.time({x=numeric(10^5); for(i in 1:10^5) x[i]=runif(1)})
##    user  system elapsed
##    1.24    0.00    1.26
```

Usuwając pętlę skrócimy czas wykonywania 6000 razy.

```
system.time({ x=NULL; x = runif(10^5) })
##    user  system elapsed
##    0.02    0.00    0.01
```




**Rysunek 2.9:** Przykład użycia pakietu profr do wizualizacji danych o czasie działania. Na osi X prezentowany jest czas w sekundach począwszy od rozpoczęcia profilowania, każdy z prostokątów odpowiada za czas działania kolejnych funkcji w tym również funkcji wywoływanych wewnętrznie na kolejnych poziomach zagnieżdżenia.

Poza wymienionymi powyżej narzędziami, do profilowania i wizualizacji informacji z profilowanego kodu można wykorzystać pakiety `proftools` (wykorzystuje pakiet graficzny `Rgraphviz` do graficznego przedstawienia wyników) oraz `profr` (wykorzystuje pakiet graficzny `ggplot2`). Poniżej przedstawiamy przykładowe użycie pakietu `profr`. Wykres, będący wynikiem tego kodu jest przedstawiony na rysunku 2.9. Wyniki profilera są konwertowane do postaci rozpoznawalnej przez przeciążoną funkcję plot.

Włączamy proces profilowania wykonania kodu. Pierwsza funkcja trwa około 25 sek, druga funkcja trwa około 12 sek.

```
Rprof("profiler.out", interval = 0.01)
tmp <- table(outer(1:N, 1:N, "*") %% 10)
tmp <- sort(rnorm(10^7))
Rprof(NULL)
```

Aby użyć pakietu `profr` należy przetransformować wyniki profilera do innej postaci.

```
out <- parse_rprof("profiler.out")
## Read 1782 items
```

Przedstawmy graficznie czasy działania.

```
plot(out)
```

Moore's Law: Processors keep getting faster and faster. Yet our datasets get bigger and bigger and an even faster rate. So we're still waiting and waiting... Result: An urgent need for high(er) performance computing with R. Dirk Eddelbuettel, useR!2010.

### 2.9.3 Jak zwiększyć wydajność programów w R (ang. High-performance computing)

Program R powstał jako pomoc dydaktyczna do nauczania statystyki. Początkowo rozwijany był przez osoby kierujące się filozofią „więcej czasu spędza się na pisaniu i debugowaniu skryptów niż trwa ich uruchomienie". Z tych powodów programy R często nie są tak efektywne jak optymalizowane procedury w językach niższego poziomu. Język programowania R jest za to znacznie bardziej przyjemny niż u konkurencyjnych „niskopoziomowych" języków programowania. Z biegiem czasu apetyt rośnie i coraz częściej użytkownicy chcą używać funkcji dostępnych w programie R na zbiorach danych o rozmiarach wielu gigabajtów wykorzystując do obliczeń wieloprocesorowe serwery. Powstało wiele pakietów oraz dodatkowych narzędzi umożliwiających poprawę wydajności programów napisanych w R, czy to przez wektoryzację obliczeń, przez wykorzystanie wielowątkowości, przez wykorzystanie procesorów kart graficznych, czy kompilację kodu R.

Poniżej przedstawimy wybrane rozwiązania pozwalające na poprawę wydajności programu R. Lista ta została opracowana na podstawie warsztatów prowadzonych na kolejnych edycjach konferencji *useR* przez Dirka Eddelbuettela. Pod adresem `http://dirk.eddelbuettel.com/papers/useR2010hpcTutorial.pdf` znajduje się prezentacja z tych warsztatów. Przedstawione w niej przykłady i linki są kopalnią cennych informacji i odnośników do innych bardziej specjalistycznych materiałów dotyczących poprawy wydajności kodu.

#### 2.9.3.1 Kompilacja w locie

Znaczne przyspieszenie czasu działania można uzyskać stosując kompilację w locie (ang. *just in time compilation*). Aby wykorzystać ten mechanizm potrzebny jest pakiet `jit` oraz specjalna wersja R nazywająca się Ra. Dla systemu operacyjnego Ubuntu program `Ra` dostępny jest w pakiecie `r-base-core-ra`, który można zainstalować poleceniem

```
apt-get install r-base-core-ra
```

Po zainstalowaniu programu `Ra` i pakietu `jit`, zamiast uruchamiać program R poleceniem `R`, należy użyć polecenia `Ra`. Pakiet `jit` działa jedynie w programie `Ra`, w zwykłej implementacji `R` nie ma żadnego efektu.

Na poniższym przykładzie pokazujemy jak wykorzystać pakiet `jit` i program `Ra`. Przykładowy kod ładuje pakiet `jit`, następnie definiuje funkcję `fib()` liczącą wartość określonej liczby z ciągu Fibonacciego oraz wywołuje tę funkcję do wyznaczenia 10 000 000 wyrazu ciągu Fibonacciego.
Testujemy bibliotekę do kompilacji w locie na czasochłonnej funkcji `fib()`.

```
library(jit)
fib <- function(N = 10) {
   # tu wlaczamy kompilacje w locie
   jit(1)
   s = c(1,1,2)
   if (N<4) return(s[N])
   for (i in 1:(N-3)) {
       s[1:2] = s[2:3]
       s[3] = s[2]+s[1]
   }
   s[3]
}
system.time(fib(10^7))
```

W powyższym kodzie na komentarz zasługuje wywołanie funkcji `jit(1)`. Wywołanie z wartością 1 oznacza, że w tym miejscu rozpoczyna się blok, od którego należy stosować kompilację w locie. Wywołanie funkcji `jit()` z wartością 0 kończy blok kompilowany w locie. Jeżeli koniec nie zostanie wskazany jawnie, będzie nim koniec funkcji w której `jit()` jest wywołana. Na powyższym przykładzie całe ciało funkcji `fib()` jest kompilowane przed wykonaniem.

Wywołanie funkcji `fib(10^7)` bez włączania pakietu `jit` kończy się po prawie 3 minutach.

```
system.time(fib(10^7))
##    user  system elapsed
## 161.120   0.100 161.372
```

Uruchomienie tego samego kodu w programie `Ra` z włączonym pakietem `jit` skraca czas działania ponad 30 razy! Wyniki dotyczą tego samego komputera i tych samych ustawień środowiskowych.

```
system.time(fib(10^7))
##    user  system elapsed
##   5.260   0.040   5.316
```

Ponad 30 krotne przyśpieszenie to rzecz godna uwagi. W powyższym przypadku przyśpieszenie było tak znaczące, ponieważ sam kod nie był napisany optymalnie. W bardzo wielu przypadkach należy się spodziewać znacznego przyśpieszenia, szczególnie na czasie działania pętli oraz instrukcji warunkowych. Więcej informacji o możliwościach optymalizacji zaimplementowanych w pakiecie `jit` można znaleźć w pliku pomocy dla funkcji `jit::jit()`.

#### 2.9.3.2 Wektoryzacja obliczeń

Porównaj z oryginalnym przykładem Dirka z prezentacji [44]. Czy potrafisz znaleźć przykład z jeszcze wyraźniejszym zyskiem w czasie działania wekoryzowanych operacji?

Jednym z najpopularniejszych mechanizmów przyśpieszania kodu programu R jest wektoryzacja obliczeń, czyli unikanie pętli i wykorzystywanie możliwie często funkcji operujących na całych wektorach lub macierzach. Bardzo pomocne w takich sytuacjach są funkcje `apply()`, `outer()`, `sapply()`, itp. opisane w tabeli 2.7. Poniżej pokażemy ile czasu można zaoszczędzić korzystając z wektoryzowanych obliczeń.

Jako przykład wykorzystamy obliczenia dla następującego problemu. Jakie jest prawdopodobieństwo, że iloczyn dwóch liczb całkowitych losowo wybranych z przedziału od 1 do 2000 kończy się cyfrą $k \in \{0, \ldots, 9\}$? Aby policzyć to prawdopodobieństwo wyznaczymy iloczyny wszystkich par liczb z przedziału 1...2000 i następnie policzymy częstości występowania różnych cyfr na ostatniej pozycji.

Przedstawimy cztery rozwiązania. Pierwsze rozwiązanie wykorzysta dwie pętle i dynamicznie tworzony wektor, czyli najgorsze z możliwych podejść.

```
N <- 2000
system.time({
  wek <- c();
  z <- 0;
  for (i in 1:N)
     for (j in 1:N)
         wek[z <- z+1] <- (i*j) %% 10;
  table(wek)})
##     user  system elapsed
##  1562.30    0.29 1562.59
##
```

Najpoważniejszym błędem jest brak wstępnej alokacji pamięci dla wektora wek. Dynamicznie tworzony wektor wek powoduje nieustanne kopiowanie całego wektora z jednego miejsca pamięci w drugie, co jest bardzo czasochłonne. W poniższym przykładzie uprzednio zaalokujemy pamięć dla całego wektora, co przyśpieszy działanie o dwa rzędy.

```
system.time({
  wek <- numeric(N^2);
  z <- 0;
  for (i in 1:N)
     for (j in 1:N)
         wek[z <- z+1] <- (i*j) %% 10;
  table(wek)})
##    user  system elapsed
##   25.46    0.09   25.74
```

Kolejne usprawnienie, to usunięcie pętli. Jedną pętlę można zastąpić użyciem funkcji `apply()` lub `sapply()`, dwie możemy zastąpić użyciem funkcji `outer()`. Zobaczmy jak to wpłynie na czas działania.

```
system.time(table(outer(1:N, 1:N, "*") %% 10))
##    user  system elapsed
##   12.20    0.08   12.31
##
```

Uniknięcie pętli przyśpieszyło czas działania dwukrotnie. Kolejne 12 sekund możemy zaoszczędzić używając funkcji `tabulate()` zamiast `table()`.

```
system.time(tabulate(outer(1:N, 1:N, "*") %% 10))
##    user  system elapsed
##    0.57    0.00    0.57
##
```

Dokładniejsza analiza pokaże, dlaczego funkcja `table()`, mimo że popularniejsza, jest znacznie wolniejsza od funkcji `tabulate()`. Wystarczy sprawdzić ciało funkcji `table()` by zobaczyć, że poza wyznaczeniem macierzy kontyngencji wykonywane jest wiele przygotowawczych operacji. Takich par funkcji, z których jedna wywołuje drugą po uprzednim sprawdzeniu typów lub przetworzeniu wstępnym danych, jest znacznie więcej. Np. wywołanie funkcji `lm.fit()` jest znacznie szybsze niż funkcji `lm()` a obie wyznaczają oceny współczynników w modelu liniowym. Innym przykładem jest wywołanie funkcji `.Internal(qsort(x))`, które podobnie jak `x[order(x)]` jest szybsze od wywołania `sort(x)`. Podobnych przykładów przytoczyć można więcej, np. wywołanie `sample.int(n)` jest szybsze niż `sample(1:n)`, itp.

#### 2.9.3.3 Inne pakiety i opcje poprawiające wydajność R

Pakietów pozwalających na zwiększanie wydajności programów napisanych w języku R jest wiele. Wystarczy zobaczyć listę prezentowaną na stronie `http://cran.r-project.org/web/views/HighPerformanceComputing.html`. Nie ma w tym rozdziale miejsca, by szczegółowo przedstawiać kilkadziesiąt pakietów, więc zamieszczam tylko podsumowanie wybranych.

- Znaczną poprawę wydajności można uzyskać używając wydajnej biblioteki BLAS do obliczeń algebraicznych (BLAS to skrót od ang. *Basic Linear Algebra Subprogram*). Zainstalowanie bibliotek kompilowanych pod dany procesor pozwala na znaczne przyspieszenie działania wielu funkcji. Atlas (ang. *Automatically Tuned Linear Algebra Software*) jest projektem udostępniającym przenośny i dynamicznie kompilowany BLAS. Dla konkretnego procesora, czy to Intel, czy AMD można również znaleźć dedykowane i wewnętrznie optymalizowane biblioteki. Domyślnie biblioteka BLAS znajduje się w pliku `R/bin/Rblas.dll`, aby użyć innej wystarczy podmienić ten plik. Sama zmiana biblioteki BLAS z domyślnej na optymalizowaną pod dany procesor może przyśpieszyć działania operacji algebraicznych nawet kilkukrotnie

- Poprawę wydajności programów R można również uzyskać rozpraszając obliczenia. Możemy rozważyć trzy rodzaje rozproszenia.
  - Dysponujemy klastrem obliczeniowym składającym się z wielu komputerów, wykorzystującym protokół `MPI` do komunikacji. Aby zaprząc wszystkie komputery z klastra do pracy można wykorzystać pakiety `Rmpi`, `snow` (popularny i bardzo łatwy w użyciu), `taskPR`, `slurm` lub `biopara`.
  - Dysponujemy kartą graficzną z szybkim procesorem GPU (ang. *graphics processing unit*). Możemy z poziomu programu R wykorzystać biblioteki lub funkcje optymalizowane dla takiego procesora. W pakietach `cudaBayesreg` i `gputools` znaleźć można algorytmy algebraiczne i statystyczne wykorzystujące GPU. Przykładowo regresja liniowa wykonywana na procesorze GPU z użyciem `gputools` może być kilkukrotnie szybsza niż wykonana na procesorze CPU (różnica w czasie wykonania zależy od konkretnego CPU i GPU). Aktualnie trwają też prace nad oprogramowaniem dla programu R interfejsu do OpenCL. Taki interfejs pozwalałby pisać programy w pełni wykorzystujące możliwości GPU.
  - Dysponujemy wielordzeniowym procesorem i chcemy by obliczenia wykorzystywały wszystkie rdzenie. Do tego celu zostały stworzone np. pakiety `multicore` (nie dla Windowsa) oraz `pnmath0`. Zwykła instalacja programu R działa jako jeden wątek na jednym rdzeniu, jeżeli mamy procesor z ośmioma rdzeniami to możemy przyśpieszyć działania łatwych do zrównoleglenia poleceń do ośmiu razy. Wymienione pakiety wykorzystują wiele rdzeni w odmienne sposoby, warto więc sprawdzić który z pakietów będzie bardziej wydajny w konkretnej sytuacji.
- Wąskim gardłem wydajności mogą być operacje przeprowadzane pamięci na dużych zbiorach danych, które nie mieszczą się w dostępnej pamięci RAM. W takich przypadkach poprawę wydajności można uzyskać stosując pakiety pozwalające na przechowywanie części danych poza pamięcią RAM przydzieloną dla programu R. Do tego celu można wykorzystać pakiet `ff` (mapuje obiekty na pliki na dysku) lub `bigmemory` (mapuje obiekty na obszary pamięci poza programem R).

> The Huli of Papua New Guinea use '15' to mean a very large number and '15 times 15 samting (something)' to mean something close to infinity.
>
> David Whiting
> R-help (April 2004)

- Jeżeli dużo pracujemy z dużymi obiektami typu `data.frame` to możemy nawet nie wiedzieć, że nawet najprostsze operacje wymagają utworzenia kilku kopii tych obiektów, przez co są dosyć wolne. Często można kilkudziesięciokrotnie przyspieszyć wykonanie naszego programu, jeżeli zamiast obiektów klasy `data.frame` użyjemy klasy `data.table`. Ta klasa umożliwia definiowanie indeksów na kolumnach, sprawniej obsługuje pamięć, unika nadmiernego kopiowania obiektów, dzięki czemu można znacznie przyspieszyć prace ze zbiorami danych w programie R. Ceną jest inny sposób korzystania z tych obiektów i specyficzna składnia poleceń operujących na obiektach `data.table`.

- Kolejnym sposobem przyśpieszenia kodu jest przepisanie fragmentów kodu R do języków kompilowanych, np. Fortrana, C lub C++. Wiele pakietów pozwala na łatwe wstawienia kompilowanego kodu. Do najpopularniejszych należą pakiety `Rcpp` (bardzo dobre wsparcie i dokumentacja) oraz `RcppArmadillo` pozwalający na wykorzystywanie bibliotek algebraicznych w C. Można też osadzać kod R wewnątrz aplikacji napisanych w C++, używając np. pakietu `RInside`.

### 2.9.4 Przydatne funkcje systemowe

Poniżej przedstawimy kilka przydatnych funkcji, które nie pasowały tematycznie do żadnego z poprzednich podrozdziałów. Zacznijmy od funkcji `system()` o deklaracji

```
system(command, intern=FALSE, ignore.stderr=FALSE, wait=TRUE,
  input=NULL, show.output.on.console=T, minimized=F, invisible=T)
```

Funkcja ta powoduje wywołanie w systemie operacyjnym komendy `command`. Może być to komenda systemowa, taka jak `dir`, ale możemy też w ten sposób uruchomić dowolny program z wybranymi parametrami.

Argumenty funkcji `system()` pozwalają określić czy wynik wykonania polecenia ma być przekazany do środowiska R (`intern=TRUE`), czy środowisko ma czekać na wykonanie się polecenia (`wait=TRUE`), czy też ma uruchomić je asynchronicznie (`wait=FALSE`), czy okno z wykonywaniem poleceniem ma być wyświetlone na ekranie (`invisible=FALSE`, okno to staje się oknem aktywnym) oraz czy wyniki wykonywania polecenia mają być wyświetlane (`show.output.on.console=TRUE`). Argumentem `input` można określić wektor łańcuchów znaków, które zostaną przesłane jako strumień wejściowy (`stdin`) do wykonywanego polecenia.

Ścieżka do pliku `firefox.exe` jest otoczona apostrofami, ponieważ zawiera spacje. Dzięki apostrofom, system operacyjny wie, że dany napis jest jeszcze ścieżką a nie opisem argumentów.

Przykładowo poniższe wywołanie uruchomi w przeglądarce Firefox stronę domową projektu R.

```
system('"c:/Program Files/Mozilla Firefox/firefox.exe"
            -url cran.r-project.org', wait = FALSE)
```

Można przytoczyć wiele przykładów, w których przydaje się funkcja `system()`. Poprzestaniemy tutaj na jednym, wykorzystującym tę funkcję do automatycznego kompilowania pliku w formacie LaTeXdo pliku `.pdf` (zobacz rozdział poświęcony pakietowi `Sweave`). Prześledźmy poniższą sekwencję poleceń.

```
Sweave("przyklad.Snw")
system("pdflatex przyklad.tex")
suffix = format(Sys.time(),".%d.%b.%Y.%H.%M.%S")
system(paste("cmd /C ren przyklad.pdf przyklad",suffix,".pdf",
       sep=""), show.output.on.console=T)
```

Korzystamy z Sweave do wyprodukowania pliku `przyklad.tex`. Poleceniem `pdflatex` kompilujemy plik LaTeX'owy. Korzystamy z polecenia systemowego `ren` do zmiany nazwy pliku `przyklad.pdf`. Dodajemy sufix określający datę i czas powstania, dzięki temu kolejne kompilacje nie nadpiszą poprzednich wyników.

**Tabela 2.16:** Inne przydatne funkcje systemowe z pakietu base.

| | |
|---|---|
| `shell.exec()` | Uruchamia wskazany plik znajdujący się na lokalnym dysku lub wskazany przez adres URL, w programie skojarzonym z danym plikiem. |
| `shell()` | Uruchamia komendę w systemie operacyjnym. Działanie identyczne z poleceniem `system()`, choć to ponoć bardziej intuicyjny interfejs. Sposób podawania argumentów do obu funkcji wygląda praktycznie tak samo. |
| `browseEnv()` | Ta funkcja przygotowuje stronę HTML z opisem środowiska oraz przestrzeni nazw, w której pracujemy oraz otwiera tę stronę w domyślnej przeglądarce www. |

## 2.10 Zadania do części „pazuRrry"

W tym rozdziale przedstawiamy zbiór zadań do samodzielnego wykonania. Zadania zostały podzielone na trzy poziomy trudności, oznaczane liczbą liter `R`. Pod adresem `http://www.biecek.pl/R/odpowiedzi.R` znajdują się przykładowe rozwiązania poniższych zadań. Zapraszamy czytelników do zgłaszania własnych ciekawych rozwiązań i/lub zadań.

R Zadanie 2.1

Odczytaj ramkę danych z zadania 1.13. Następnie zamień dane liczbowe z kolumny `Wiek` na zmienną czynnikową, dzieląc pacjentki na 3 grupy: o wieku do 45 lat, o wieku powyżej 55 lat i o wieku pośrednim. Poziomy tej zmiennej powinny nazywać się następująco: `"wiek <45"`, `"45<= wiek <=55"`, `"wiek >55"`. Następnie wyświetl macierz kontyngencji dla tej zmiennej i dla pary zmiennych wieku oraz dla płci. Dodaj do macierzy sumy brzegowe. Wyświetl płaską macierz kontyngencji dla trójki zmiennych czynnikowych, dwóch powyższych i jeszcze zmiennej `WIT`.

RR Zadanie 2.2

Pod adresem `http://www.biecek.pl/R/dane/imieniny.txt` znajduje się plik tekstowy z imionami i datami imienin dla kolejnych imion. Plik jest w dosyć kłopotliwym formacie, mianowicie w każdym wierszu w pierwszej pozycji znajduje się imię, a po nim występują daty imienin. Wszystkie te pola rozdzielone są spacją. Ponieważ jednak różne imiona mają różne liczby imienin dane te nie są w postaci tabelarycznej. Odczytaj dane tak, by każdy wiersz był jednym elementem (można np. za separator wskazać `;` nie występuje on w tym pliku, cała linia zostanie więc traktowana jako jeden element). Sprawdź ile imion znajduje się w tym pliku z danymi.

Pamiętaj przy tym, że domyślnie pola tekstowe konwertowane są na zmienne czynnikowe.

RRR Zadanie 2.3

Po odczytaniu danych z zadania 2.2 użyj `strsplit()` i `sapply()`, by z odczytanych danych wydobyć tylko informacje o imionach. Zlicz liczby znaków w kolejnych imionach i wyznacz macierz kontyngencji opisującą ile imion ma określoną długość. Sprawdź, które imię ma najwięcej znaków i które imiona mają najmniej znaków. Większość imion żeńskich kończy się literą `'a'`,

wykorzystując tę informację zlicz liczbę żeńskich imion. Sprawdź ile liter rozpoczyna się literą `A`, ile `B`, ile `C` itp. Sprawdź, które imiona kończą się sufixem `anna`. Polskie litery występujące w imionach zamień na odpowiedniki łacińskie, np. ą na `a`, ź na `z` itp. Sprawdź, ile imion zawierało polskie litery.

Od tej reguły są wyjątki np. żeńskie imię Beatrycze i męskie Bonawentura. Dla uproszczenia zapomnijmy o wyjątkach.

RR Zadanie 2.4

Po odczytaniu danych z powyższego zadania sprawdź, kto ma imieniny 30 października. Wyświetl te imiona w porządku leksykograficznym. Sprawdź, które imię ma najczęściej imieniny. Sprawdź, w którym dniu roku najwięcej imion ma imieniny. Sprawdź, w którym dniu miesiąca a następnie, w którym miesiącu najwięcej imion ma imieniny.

Na bazie wszystkich imion zrób analizę używalności poszczególnych liter. Która litera jest najpopularniejsza? Dlaczego?

R Zadanie 2.5

Odczytaj ramkę danych z zadania 1.13. Następnie używając funkcji `by()` wyświetl podsumowanie zmiennej `Wiek` osobno dla grupy `WIT=brak` i dla grupy `WIT=obecny`.

R Zadanie 2.6

Wyznacz wyznacznik, wartości własne oraz wektory własne macierzy:

$$\begin{bmatrix} 1 & 5 & 3 \\ 2 & 0 & 5 \\ 1 & 2 & 1 \end{bmatrix}.$$

RR Zadanie 2.7

Używając funkcji `outer()` zbuduj i wyświetl na ekranie tabliczkę mnożenia liczb od 1 do 10.

R Zadanie 2.8

Odczytaj ramkę danych z zadania 1.13. Następnie wyznacz histogram dla zmiennej `Wiek` i zapisz go do pliku `hist.pdf` w wymiarach $5 \times 5$ cali.

RRR Zadanie 2.9

Pod adresem `http://money.pl/` można odczytać aktualne wartości indeksów giełdowych. Wczytaj zawartość tej strony do programu R, a następnie wyciągnij z niej dane o nazwie i wartościach indeksów.

Zauważ, że w treści HTML tabela z nazwami i wartościami indeksów rozpoczyna się od linii `<table id="tgpw" class="tabela">`.

RR Zadanie 2.10

Używając profilera spróbuj przyśpieszyć poniższy fragment kodu. Sprawdź, które instrukcje są czasochłonne i zastąp je szybszymi. Następnie sprawdź jak bardzo udało Ci się przyśpieszyć ten kod.

Wskazówka: Funkcja `cor()` wywołuje masę dodatkowych funkcji, a tylko jedną niezbędną, czyli `Internal()`.

```
Rprof("profiler.out", interval = 0.01, memory.profiling=TRUE)
wyniki = NULL
for (i in 1:10000) {
        wektorX <- rnorm(20)
        wektorY <- rnorm(20)
        wyniki[i] <- cor(wektorX, wektorY)
}
Rprof(NULL)
summaryRprof("profiler.out", memory="both")
```

RR Zadanie 2.11

Napisz funkcję `test()`, której argumentem będzie komenda do sprawdzenia. Funkcja `test()` powinna wykonać daną komendę. Jeżeli zakończy się ona błędem to funkcja `test()` powinna wypisać na ekranie napis `KLAPA`, a jeżeli wykonanie będzie bezbłędne, to powinien pojawić się napis `SUKCES`. Przykładowe wywołanie `test(cor(1))` powinno spowodować wyświetlenie napisu `KLAPA`.

R Zadanie 2.12

Znajdź miejsca zerowe wielomianu $x^3 - 3x^2 - x + 3$. Znajdź najmniejszy wspólny dzielnik tego wielomianu i $x^3 - 12x^2 - x + 12$.

R Zadanie 2.13

Używając funkcji `head()` i `by()` dla zbioru danych z zadania 1.13 wyświetl trzy pierwsze wiersze danych dla mężczyzn i trzy pierwsze wiersze danych dla kobiet.

RR Zadanie 2.14

Jeżeli argumentem funkcji `diag()` jest macierz, to wynikiem jest wektor wartości z przekątnej. Napisz funkcję `diag2()` przyjmującą dwa argumenty. Pierwszym będzie macierz danych a drugim liczba. Wynikiem będzie przekątna przesunięta o wartość drugiego argumentu. Przykładowo, jeżeli drugi argument będzie miał wartość `0`, to wynikiem będzie główna przekątna, jeżeli będzie miał wartość 1, to wynikiem będą elementy położone bezpośrednio nad główną przekątną, jeżeli wartość 2, to wyniki położone dwa wiersze ponad główną przekątną itp.

RR Zadanie 2.15

Napisz funkcję wyświetlającą nazwy 10 zmiennych zajmujących najwięcej pamięci w przestrzeni roboczej R.

R Zadanie 2.16

Pod adresem `http://www.biecek.pl/R/dane/daneBioTech.xls` znajduje się plik z danymi w formacie programu Excel. Skopiuj ten plik na dysk twardy, a następnie odczytaj dane z obu zakładek.