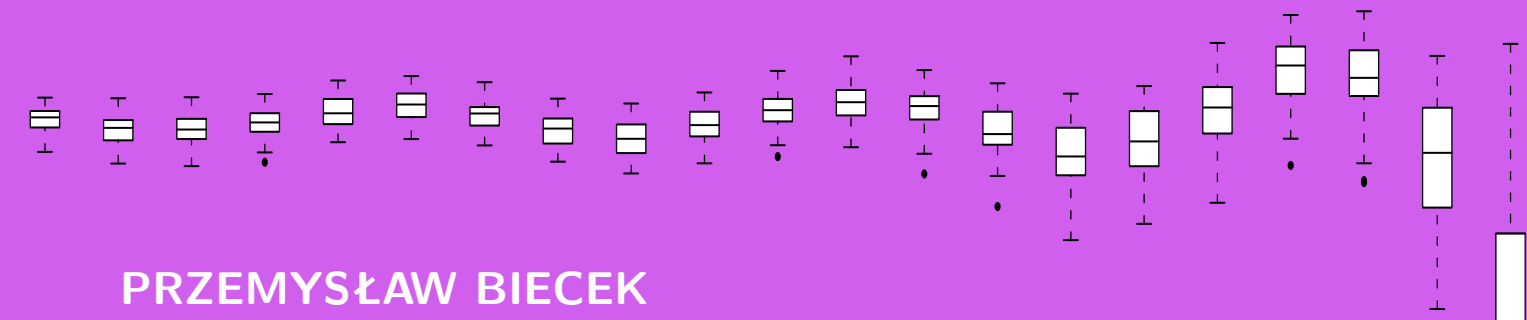
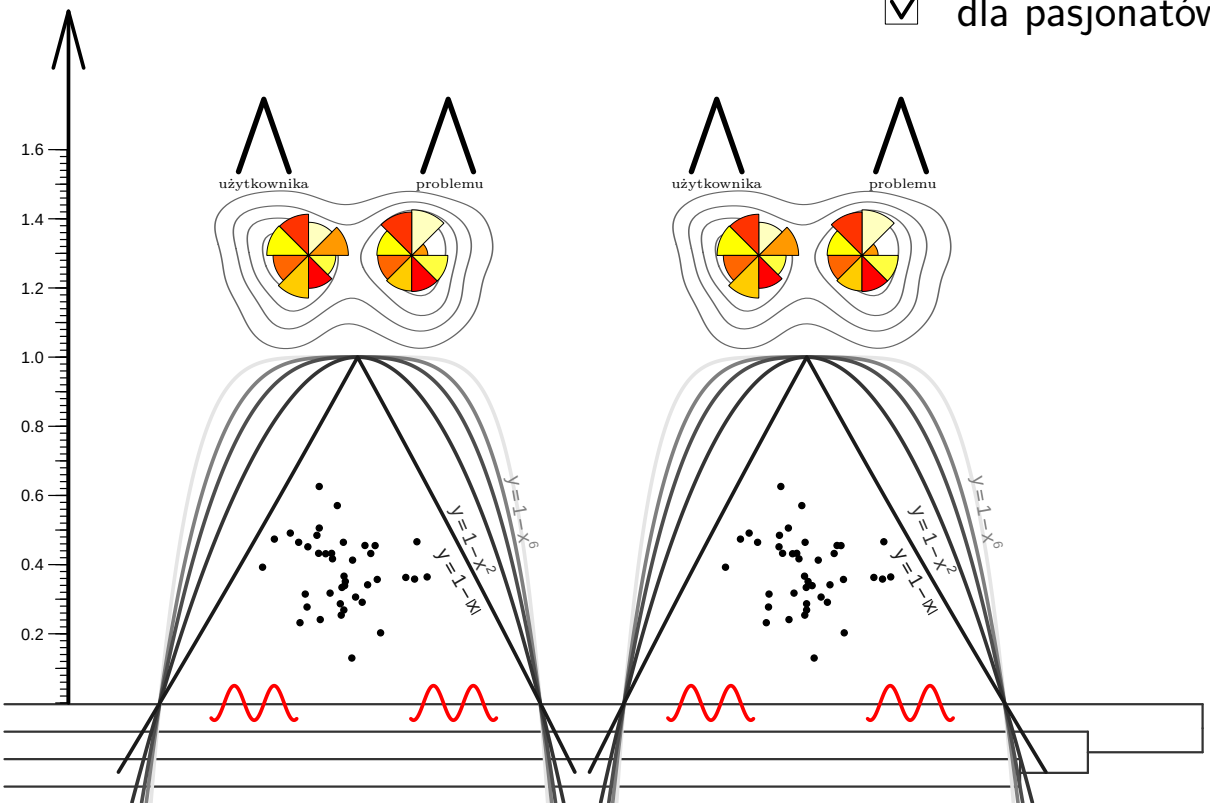


PRZEMYSŁAW BIECEK

Przewodnik po pakiecie

R

- ☑ dla żółtodziobów
- ☑ dla zawodowców
- ☑ dla pasjonatów



STATYSTYKA ANALIZA DANYCH GRAFIKA PROGRAMOWANIE

Recenzent
Dr hab. Jan Mielniczuk
Instytut Podstaw Informatyki PAN

Projekt okładki i skład
Przemysław Biecek

Wnioski, skargi i zażalenia kierować do
Przemysław Biecek
http://www.biecek.pl
Instytut Matematyczny Polskiej Akademii Nauk
Zakład Genomiki Wydziału Biotechnologii Uniwersytetu Wrocławskiego

Książka została przygotowana, aby ułatwić poznanie i codzienna pracę z pakietem R. Przyda się ona tym wszystkim, którzy w pracy lub w szkole zajmują się analizą danych. Książka może być wykorzystana, jako pomoc w nauce pakietu R. Może być również wykorzystana, jako encyklopedyczna ściągawka przydatna w codziennej pracy z tym pakietem.
Pod adresem **http://www.biecek.pl/R/** *czytelnik znajdzie dodatkowe informacje o książce, rozwiązania zadań umieszczonych w tej książce, oraz odnośniki do innych materiałów wspomagających naukę pakietu R.*

Osoby zainteresowane zakupem książki powinny skontaktować się z autorem.

Spis treści

Przeczytaj zanim kupisz

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, zaczynając 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 nazwy wszystkich argumentów graficznych) lub też chcące upewnić się, że o R wiedzą już (prawie) wszystko.

Innymi słowy, mam nadzieje, 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 jakiejkolwiek 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 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 podstawowych jak i bardzo wymyślnych), schematów, grafik itp. Część czwarta kończy się prezen-

tacją funkcji i argumentów graficznych, dzięki którym użytkownik ma pełną kontrolę nad tym co, jak i gdzie jest rysowane.

Pakiet 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 pakietem R. Ten fragment może być drukowany i kopiowany na użytek własny. Mam nadzieje, że pomoże on wielu osobom w pierwszym kontakcie z R, a także zachęci do nabycia całej książki w postaci drukowanej.

Książka ta mogła powstać wyłączenie dzięki mniejszej i 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, za co serdecznie mu dziękuję. Za cenne uwagi merytoryczne chciałbym też podziękować dr Janowi Ćwikowi i dr hab. Pawłowi Mackiewiczowi a również Grzegorzowi Hermanowiczowi i moim studentom, którzy czasem dzielili się uwagami czy wątpliwościami. Za pomoc przy wydawaniu tej książki chcę podziękować prof. dr hab. Jackowi Koronackiemu. 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 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 oraz wielu sukcesów w pracy z użyciem pakietu 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 wyróżnione. Kod w języku R oraz przykłady wyników wykonania podanych instrukcji będą przedstawiane w następujących ramkach:

```
# komentarz: mój pierwszy program

for (i in 1:10) {
        cat("Hello world !!!\n")
}
```

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

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 R. Te i więcej cytatów znaleźć można w pakiecie `fortunes`.

Fragmenty tekstu zasługujące na szczególną uwagę oraz komentarze do przedstawianego zagadnienia będą oznaczane krzywą opisaną równaniem w układzie biegunowym $G = \{(\rho, \phi) : \rho = 1 + 1/|\phi|, -\pi \leqslant \phi \leqslant \pi\}$ (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 literaturowych będziemy odnosili się następująco: [1].

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 w ostatnim załączniku, pliki z przykładowymi odpowiedziami znajduje się w Internecie pod adresem `http:\\www.biecek.pl\R\`. Pod tym adresem umieszczane będą 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 interpretator 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 is the lingua franca of statistical research. Work in all other languages should be discouraged.

Jan de Leeuw
fortune(78)

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, z pewnością nie będzie miał żadnych problemów przy opanowywaniu kolejnych pakietów.

Pierwsza wersja R została napisana przez Roberta Gentlemana i Ross 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 byli eksperci z różnych dziedzin (statystyki, matematyki, metod numerycznych oraz szeroko pojętej informatyki) z całego świata. Liczba osób rozwijających R szybko rosła, a aktualnie rozwojem R kieruje fundacja „The R Foundation for Statistical Computing" z dziesiątkami aktywnych uczestników. Ponadto w rozwój R mają wkład setki osób z całego świata publikujące własne biblioteki najróżniejszych funkcji z bardzo różnych dziedzin.

Język R był wzorowany na języku S, który został opracowany w laboratoriach Bell'a. Z tego też powodu język R jest 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 w pakiecie S polecam pozycję Briana Everitta [3]. Uzupełnieniem do pozycji literaturowych jest olbrzymia liczba stron internetowych oraz dokumentów elektronicznych szczegółowo przedstawiających rozmaite aspekty 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 R. Pojawiają się też i będą się pojawiały liczne książki poświęcone wybranym aspektom pakietu 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 podstawowe koncepty statystyki [21].

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 rad znajdziemy pod adresem [2].

R jest projektem GNU opartym o licencje GNU GPL. W uproszczeniu 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, dokumentów chm lub stron html. Aktualnie dokumentacja ta jest angielskojęzyczna, jednak trwają prace nad różnymi lokalizacjami.

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 z klawiatury lub też mogą być wykonywane jako skrypt (czyli plik tekstowy z zapisaną listą komend do wykonania). Skrypty można wykonywać niezależnie od platformy sprzętowej. 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.

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 procedur statystycznych, podsumowań i wykresów. Zdecydowanie jednak zachęcam takie osoby 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.

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

Pierwszy podrozdział zakończę przedstawieniem czterech głównych (ale nie jedynych) zalet platformy R. Dzięki tym zaletom deklasuje ona konkurencję.

- R pozwala na tworzenie i upowszechnianie pakietów zawierających nowe funkcjonalności. Obecnie dostępnych jest blisko 1000 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 dla innych.

- R pozwala na wykonywanie funkcji z bibliotek dostępnych w innych językach (C, C++, Fortran) oraz na wykonywanie funkcji dostępnych w R z poziomu innych języków (Java, C, C++ i wiele innych). Dzięki temu możemy np. znaczną część programu napisać w Javie, a R wykorzystywać jako dużą zewnętrzną bibliotekę funkcji statystycznych.

- 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 R można wykonać wykresy o wysokiej jakości, co jest bardzo istotne przy prezentacji wyników. Wykresy te już na pierwszy rzut oka wyglądają lepiej od tych przygotowanych w innych pakietach.

Panie, takie rzeczy to tylko w eRze

Jedną z niewielu rzeczy których nie można zrobić na platformie R jest cappucino.

1.3 Instalacja

Instalacja pakietu R składa się z dwóch etapów. Pierwszy, to zainstalowanie podstawowego środowiska (tzw. *base*) 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 około tysiąca pakietów! Nie ma jednak potrzeby instalowania wszystkich od razu. Z reguły 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.

1.3.1 Instalacja środowiska

Mirror to serwer, 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.

Dla większości systemów operacyjnych, w tym wszystkich dystrybucji Linuxa, Unixa, dla wersji Windowsa począwszy od Windowsa 95 a nawet dla MacOSa, pakiet R jest dostępny w postaci źródłowej oraz skompilowanej. Łatwiej oczywiście 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 mirrorów z kopiami plików instalacyjnych pakietu R znaleźć można pod adresem `http://cran.r-project.org/mirrors.html`). W większości przypadków najszybciej ściągniemy pakiet z jednego z polskich mirrorów.

Szczegółową instrukcję instalacji można znaleźć pod adresem [8], przyda się ona osobom chcącym zainstalować nietypową konfigurację R. W dalszej części będzie opisywana wersja pakietu R przygotowana dla systemu Windows. Jej najnowszą wersję (na dzień dzisiejszy 2.7.1) można ściągnąć np. z wrocławskiego serwera [7]. Aby przystąpić do instalacji należy uruchomić plik `R-2.7.1-win32.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-2.7.1`) 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 w tle chcemy wykonać jakieś większe symulacje i nie potrzebujemy interfejsu graficznego. Wybór trybu uruchomienia proponujemy oprzeć na prostej zasadzie: jeżeli nie wiesz czym te tryby się różnią, to uruchom `Rgui.exe`.

Osoby używające platformy 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.

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, 2GB RAM i tyle samo miejsca na dysku twardym (bioinformatyczne zbiory danych potrafią zajmować bardzo dużo miejsca na dysku i w RAM).

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

1.3.2 Instalacja i ładowanie pakietów

Jak już pisaliśmy, po zainstalowaniu podstawowego zbioru bibliotek platforma R ma już spore możliwości. Prawdziwa potęga kryje się w setkach dodatkowych pakietów, w których znajdują się tysiące różnych funkcji (funkcje w R pogrupowane są w pakietach/bibliotekach). Po uruchomieniu systemu R kolejne pakiety można zainstalować funkcją `install.packages(utils)`.

```
# zainstaluj pakiet Rcmdr wraz z wszystkimi pakietami wymaganymi do jego
    działania
install.packages("Rcmdr", dependencies = TRUE)
```

lub też wybierając z menu opcję `packages\install package(s)...`. Przy instalacji pierwszego pakietu R zapyta z jakiego serwera mirror chcemy skorzystać.

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(base)`.

```
# włącz pakiet Rcmdr
library(Rcmdr)
# gdyby ten pakiet nie był zainstalowany, to pojawiłby się komentarz
# Error in library(Rcmdr) : there is no package called 'Rcmdr'
```

Jak już pisaliśmy, aktualnie dostępnych jest blisko 1000 pakietów, które możemy dodatkowo zainstalować. W tym zbiorze trudno czasem odnaleźć pakiet z interesującą nas funkcjonalnością. Dlatego też, przedstawiając nowe funkcje będziemy korzystać z notacji `nazwaFunkcji(nazwaPakietu)`. Tak więc zapis `wilcox.test(stats)` będzie oznaczać, iż funkcja `wilcox.test()` znajduje się w pakiecie `stats`. Również w skorowidzu, znajdującym się na końcu książki, dla każdej wymienionej funkcji określamy w jakim pakiecie jest ona dostępna. Jeżeli znamy nazwę funkcji i chcemy

dowiedzieć się w jakim pakiecie ta funkcja się znajduje, to możemy skorzystać z funkcji `help.search(utils)`. 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 1.5.3.

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()` (w każdym o innym działaniu). Aby wskazać z którego pakietu chcemy wybrać funkcję należy użyć operatora `::`.

Jeżeli nie użyjemy tego operatora a dojdzie do kolizji nazw to środowisko R zapyta, z którego pakietu chcemy uruchomić daną funkcję. Obie poniższe linie wywołają funkcję `seq()` z pakietu `base`.

```
# oba 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)
```

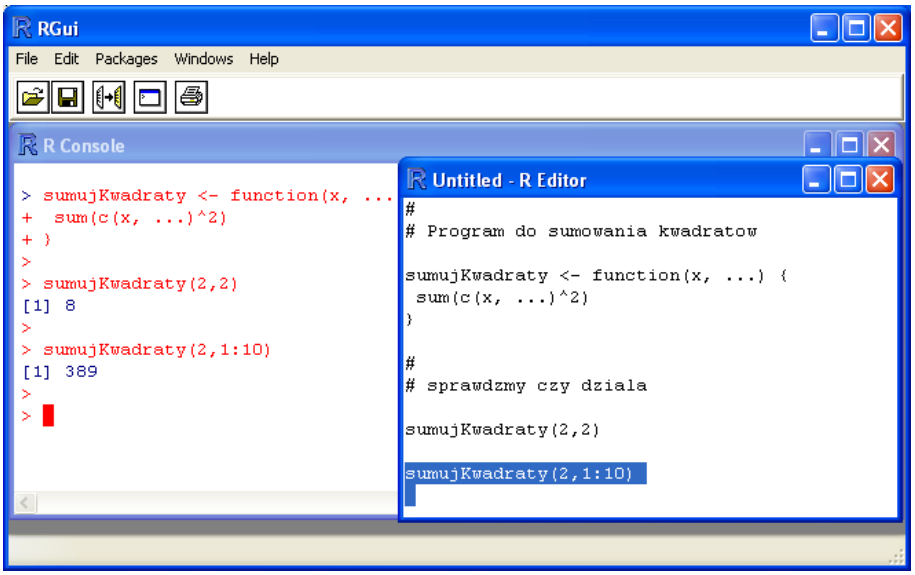
1.4 EdytoR

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 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.

Do edycji skryptów można wykorzystać dowolny edytor obsługujący pliki tekstowe (począwszy od najprostszego z możliwych, czyli programu Notatnik z systemu Windows). Po napisaniu skryptu możemy cały kod programu lub jego fragment skopiować do schowka i przekopiować do konsoli R (poprzez schowek, a więc skrótami klawiszowymi `Ctrl-C`, `Ctrl-V`). Oczywiście zamiast Notatnika możemy wykorzystać dowolny inny edytor, z którym lubimy pracować. Takie rozwiązanie jest wygodne gdy piszemy krótkie skrypty, wprowadzamy niewielkie zmiany lub gdy korzystamy z R uruchomionego zdalnie, np. na unixowym serwerze. W takich sytuacjach najczęściej nie potrzebujemy żadnych specjalistycznych edytorów.

Jeżeli chcemy uruchomić w R cały skrypt z przygotowanym kodem programu, to zamiast kopiować ten kod przez schowek możemy skorzystać z funkcji `source(base)`. Ten sam efekt wyklikamy z menu poleceniem **File/Source R code...**. Jeżeli argumentem funkcji `source()` będzie ścieżka do pliku, to cały ten plik zostanie wczytany i wykonany w R. Jeżeli argumentem będzie napis `"clipboard"`, to wykonane zostaną polecenia znajdujące się w schowku systemowym. Oba te rozwiązania są lepsze niż wklejanie kodu bezpośrednio do konsoli ponieważ ekran nie jest zaśmiecany wklejanym kodem.

O ile notatnik nie ma żadnego wsparcia do R, to minimalne wsparcie ma wbudowany w RGui edytor. Można go otworzyć poleceniem **File/New script** z menu



Rysunek 1.1: Przykładowe okno wbudowanego edytora RGui. Skrótem *Ctrl-R* można wysłać zaznaczony fragment kodu do konsoli R

(otwiera się pusty skrypt), poleceniem **File/Open script** (otwieramy istniejący skrypt do edycji) lub funkcją `edit(utils)`. Ten wbudowany edytor ma kilka udogodnień. Przykładowo po zaznaczeniu fragmentu kodu skrótem klawiszowym `Ctrl-R` kopiujemy ten fragment kodu (lub aktualną linię kodu, jeżeli nic nie jest zaznaczone) do konsoli R. Typowe okno tego edytora przedstawiamy na rysunku 1.1. Jest to wygodne narzędzie do tworzenia krótkich programów jak i wprowadzania drobnych modyfikacji do już napisanych skryptów.

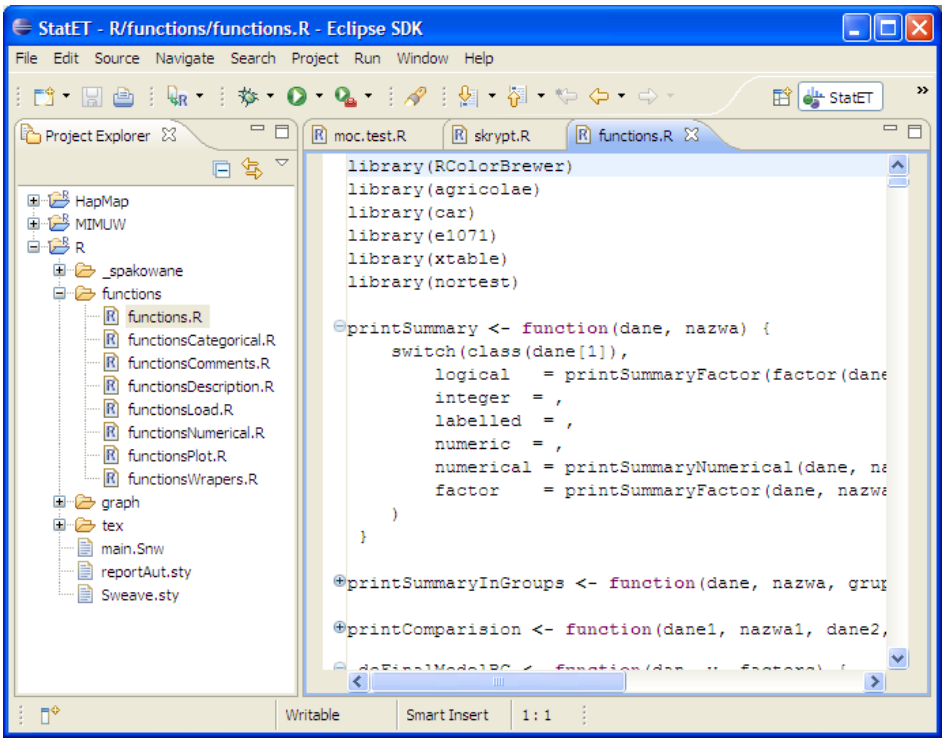
Funkcja *fix()* umożliwia zmianę wartości dowolnego obiektu, także na edycje ciała funkcji!

Jeżeli podczas pracy z R zachodzi potrzebna zmodyfikowania wartości jakiegoś obiektu (np. funkcji lub tabeli danych), to drobne modyfikacje wygodnie jest wykonywać używając funkcji `fix(base)`. Powoduje ona otwarcie okna edytora R z aktualną wartością obiektu będącego argumentem tej funkcji. Po zakończeniu edycji tej wartości, zamykając okno edytora zmieniana jest również wartość danego obiektu w środowisku.

Aby wygodnie pracować przy dużych projektach, gdzie kod rozmieszczony jest w wielu plikach, potrzebujemy lepszego wsparcia do R. Wiele popularnych edytorów zawiera makra lub pliki definicji pozwalające na podstawowe wsparcie, takie jak np. kolorowanie składni. Osoby używające edytora Emacs z pewnością ucieszy informacja, że do Emacsa przygotowano wtyczkę pozwalającą na edycje skryptów R. Wtyczka nazywa się ESS (skrót od Emacs Speaks Statistics), wspiera ona edycję i uruchamianie skryptów dla wielu pakietów statystycznych w tym tych z rodziny języka S. Więcej informacji o tej wtyczce można znaleźć pod adresem [9].

Dla programistów programujących w Javie lub C++ dobrym wyborem będzie platforma Eclipse [10]. Jest to platforma do programowania w Javie, jednak kolejne wersje wspierają też wiele innych języków programowania, a dostępna wtyczka (plug-in) „StatET" [11] umożliwia wygodną współpracę Eclipse ze środowiskiem R. Środowisko Eclipse zostało napisane w Javie, dzięki temu można je uruchamiać zarówno pod Linuxem jak i pod Windowsem.

Przykładowe okno edytora Eclipse z zainstalowaną wtyczką StatET przedstawione jest na rysunku 1.2. Co może być przydatne w Eclipse możemy mieć jednocześnie otwarte projekty w R, Javie czy innych językach programowania.

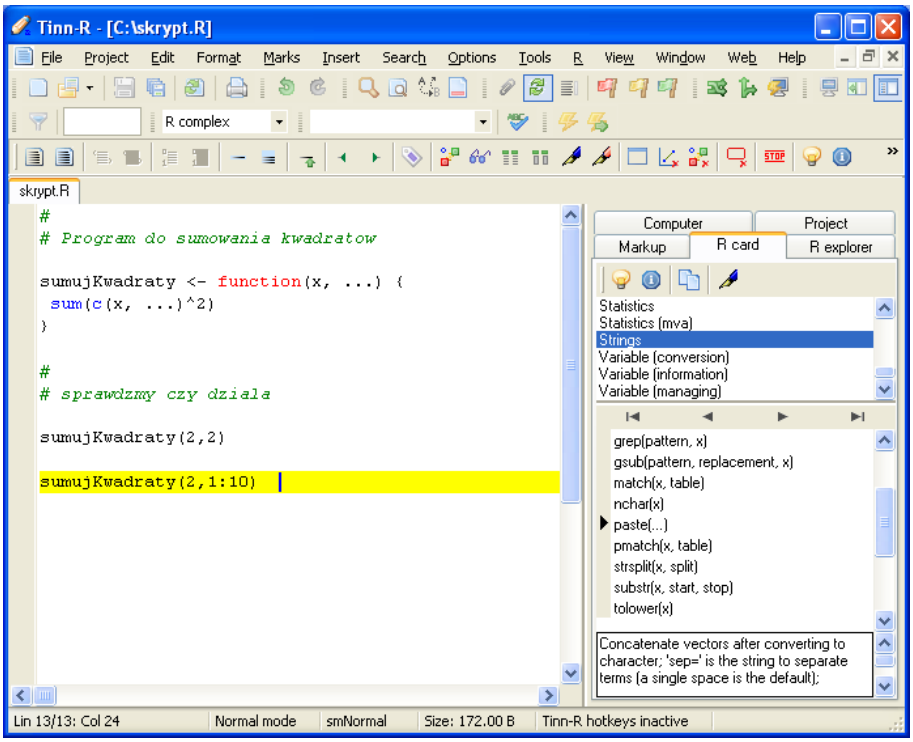


Rysunek 1.2: Przykładowe okno edytora Eclipse. Po lewej stronie jest okno projektów. Po prawej stronie widać przykład zwiniętych funkcji, zwiększa to czytelność kodu

Wtyczka „StatET" oferuje programistom R wiele usprawnień ułatwiających tworzenie dużych projektów i pracę z wieloma plikami, wystarczy wspomnieć o najważniejszych udogodnieniach:

- zarządzanie wieloma plikami/projektami.
- podświetlanie składni,
- domykanie otwartych nawiasów, cudzysłowów, wraz z inteligentnym zaznaczaniem zawartości (dwukrotne kliknięcie we wnętrze nawiasu, zaznacza cała zawartość nawiasu),
- zwijanie ciała funkcji, bardzo wygodne jeżeli piszemy dużo funkcji,
- inteligentne wstawianie wcięć połączone z rozpoznawaniem składni (czyli nowe wcięcie dodawane jest w pętlach, funkcjach itp),
- możliwość automatycznego wysyłana całego skryptu lub fragmentu kodu do konsoli R (wykorzystywana jest zintegrowana konsola R (Rterm), działa to jak na razie jedynie pod systemem Windows).

Minusem, o którym trzeba uczciwie powiedzieć, jest duża objętość platformy Eclipse. Ta platforma to prawdziwy kombajn, profesjonalna platforma programistyczna z bardzo zaawansowanymi możliwościami, dlatego też jej podstawowa instalacja wymaga przynajmniej 220MB na dysku twardym. Ponieważ Eclipse napisane jest w Javie to również intensywnie wykorzystuje pamięć operacyjną. Na szczęście ta uciążliwość odczuwalna będzie jedynie na starszych komputerach. W zamian otrzymujemy wiele rozwiązań przydanych w pracy grupowej (np. wsparcie do CVS) oraz w zarządzaniu dużymi fragmentami kodu.



Rysunek 1.3: Przykładowe okno edytora Tinn-R. Po prawej stronie widać zakładkę R Card z pogrupowaną listą przydatnych funkcji wraz z krótkim ich opisem

Innym, bardzo popularnym edytorem jest Tinn-R. To nieduży (w porównaniu do platformy Eclipse) edytor ze wsparciem dla R oraz kilku innych języków. Przykładowe okno tego edytora jest przedstawione na rysunku 1.3. Tinn-R powstał po to, by umożliwić łatwą współpracę z R, jest też z R najsilniej zintegrowany. Dokładniejszy opis jego możliwości znaleźć można na stronie internetowej [12]. Warto wymienić kilka udogodnień, które ten edytor zawiera:

- podświetlanie składni,
- możliwość automatycznego wysyłania całego skryptu lub fragmentu kodu do R (poprzez Rgui),
- zakładka R Card, z listą użytecznych funkcji opatrzonych krótkimi opisami,
- baza podpowiedzi (tipsów), pisząc jakieś polecenie skrótem klawiszowym `Ctrl-D` wyświetlamy podpowiedź informującą o liście argumentów, opisie działania itp.,
- uzupełnianie kodu, uzupełniane są nazwy zmiennych, funkcji i innych obiektów z przestrzeni roboczej R,
- monitoring listy obiektów ze środowiska R, możemy na bieżąco kontrolować jakie obiekty znajdują się w pamięci, mamy też możliwość podglądnięcia oraz zmiany ich wartości.

Z opisanych powyżej edytorów największe wsparcie dla platformy R ma Tinn-R. Z uwagi na wbudowaną pomoc zdecydowanie polecam go osobom początkującym oraz średniozaawansowanym. Do pracy z dużymi projektami polecam Eclipse.

1.5 Startujemy

Zakładamy, że czytelnik ma już zainstalowany na dysku pakiet 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 pakietu R, czas na pierwsze jego uruchomienie. W systemie Windows najlepiej uruchomić plik `Rgui.exe` z katalogu `bin`. Uruchamia on R z wbudowanym interfejsem graficznym. Platformę R można uruchomić również w trybie wsadowym lub trybie tekstowym, ale to jest temat, który omówimy w rozdziale 2.2. 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 XP, dla innych systemów nazwy funkcji i argumentów są takie same.

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 1.4.

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.

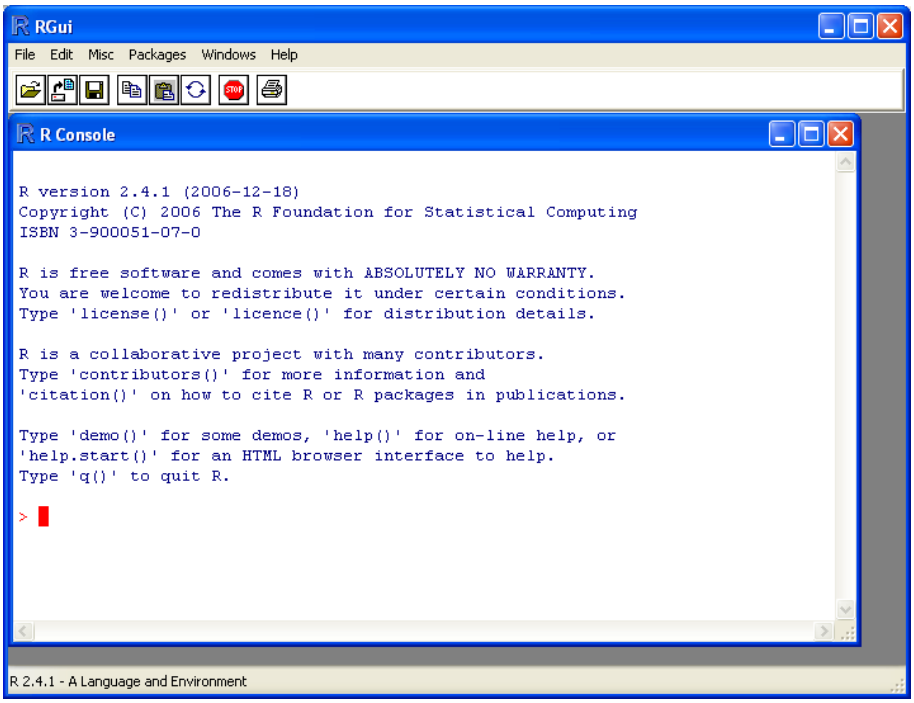
Pierwsze polecenie, które warto przećwiczyć to

```
q()
```

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

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

Jeżeli nie mamy już problemu z zamykaniem platformy R, to spróbujmy znaleźć motywacje do dalszej nauki. Dla wielu pakietów oraz funkcji dostępnych w 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 1.4: Okno powitalne, otrzymane po uruchomieniu pliku Rgui.exe

```
# poniższym poleceniem uruchamiamy graficzny interfejs, pozwalający na
    wyklikanie większości podstawowych statystyk
library(Rcmdr)

demo(persp)       # prezentacja funkcji persp, rysowanie rzutów
demo(graphics)    # prezentacja pakietu graphics, funkcji graficznych
demo(Japanese)    # znaki Kanji
library(lattice)
demo(lattice)     # prezentacja pakietu lattice
library(rgl)
demo(rgl)         # prezentacja pakietu rgl
demo(lm.glm)      # prezentacja wykresów diagnostycznych dla uogólnionych
    modeli liniowych
```

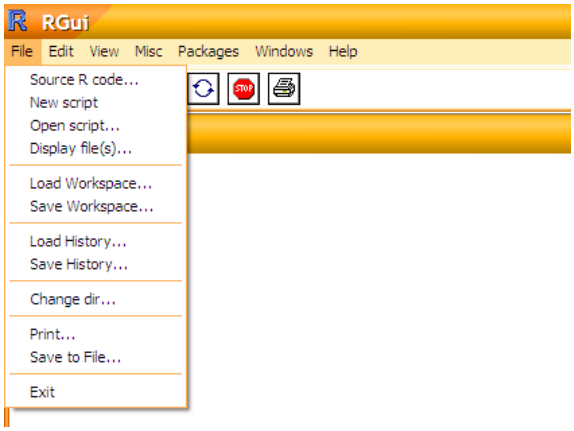
Teraz powinniśmy być już wystarczająco zmotywowani. Kolejny podrozdział przedstawia poszczególne opcje menu w okienkowej wersji R.

1.5.2 Przegląd opcji w menu

Być może będzie trochę nudno, ale warto choć przejrzeć listę pozycji w menu. Znajomość niektórych opcji może nam zaoszczędzić sporo czasu.

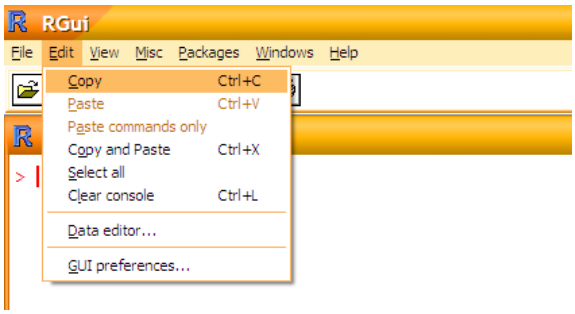
W menu dostępnym dla okienkowej wersji R jest sporo opcji. Zawartość menu zależy od tego, czy aktywne jest okno z konsolą do wpisywania poleceń, okno graficzne (okno, w którym R wyświetla wyniki funkcji graficznych), okno edytora, czy okno z pomocą. Poniżej przedstawiamy opcje dla menu widocznego gdy aktywna jest konsola poleceń lub okno graficzne. Menu dostępne gdy aktywne są inne okienka ma podobne opcje. Zacznijmy od menu dla konsoli poleceń.

- **File**



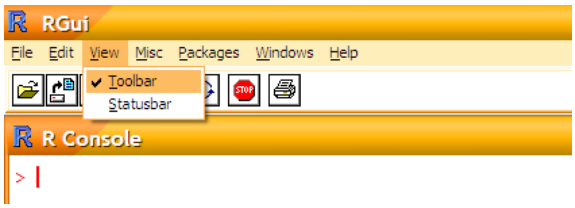
  - **Source R code...**
    Tym poleceniem możemy wskazać plik tekstowy z listą komend w języku R do uruchomienia w konsoli. Podobny efekt można uzyskać funkcją `source()`.
  - **New script**
    Polecenie otwiera wbudowany edytor skryptów R do edycji nowego pliku.
  - **Open script...**
    Polecenie otwiera wskazany plik R w wbudowanym edytorze skryptów R.
  - **Display file(s)...**
    Polecenie wyświetla zawartość wskazanych plików (każdy otwiera się w innym okienku).
  - **Load Workspace...**
    Odczytuje zapisany obszar roboczy. Terminem obszar roboczy określa się informacje o wszystkich obiektach, znajdujących się aktualnie w pamięci R. Odczytując zapisany obszar roboczy wracamy do zapisanego stanu wszystkich obiektów.
  - **Save Workspace...**
    Zapisuje obszar roboczy do wskazanego pliku.
  - **Load History...**
    Odczytuje informacje o historii wykonywanych poleceń.
  - **Save History...**
    Zapisuje informacje o historii wykonywanych poleceń.
  - **Change dir...**
    To polecenie służy do zmiany aktualnego katalogu.
  - **Print...**
    Drukuje zawartość konsoli.
  - **Save to File...**
    Zapisuje do pliku tekstowego zawartość konsoli.
  - **Exit**
    Zamyka program R (zazwyczaj pytając uprzednio o to, czy nie zapisać obszaru roboczego).

- **Edit**

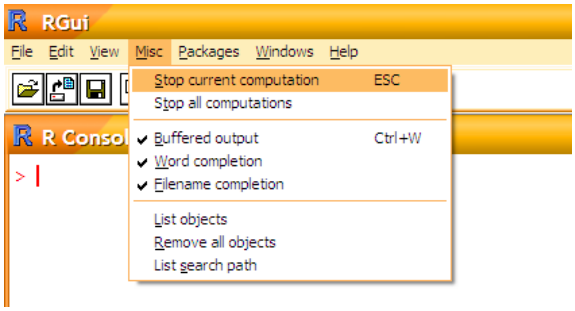


  - **Copy, Paste, Paste commands only, Copy and Paste, Select all**
    Standardowe (windowsowe) operacje do kopiowania i wklejania informacji do i ze schowka.
  - **Clear console**
    Polecenie czyści okno konsoli R, to bardzo przydatna opcja. Można ją wywołać skrótem klawiszowym `Ctrl+L`.
  - **Data Editor**
    To polecenia otwiera wbudowany edytor do danych w postaci tabelarycznej. Można w nim edytować macierze i obiekty typu `data.frame`, znajdujące się w pamięci R.
  - **GUI Preferences**
    To polecenie pozwala na zmianę różnych właściwości (głównie wyglądu) konsoli R.

- **View**
  W tym menu można włączyć lub wyłączyć wyświetlanie paska stanu oraz paska z narzędziami.



- **Misc**



  - **Stop current computation**
    Ta komenda pozwala przerwać aktualne wykonywane polecenie, pętlę lub instrukcję warunkową.
    Skrót klawiszowy to `ESC`.

Przydatna opcja, gdy już nie chcemy czekać na wynik, który liczy się dłużej niż się spodziewaliśmy.

  - **Stop all computations**
    Przerywa wykonywanie wszystkich poleceń, które są wykonywane lub czekają w kolejce na wykonanie. Jeżeli kopiujemy do konsoli więcej poleceń niż jedno (np kilka pętli), to klawiszem `ESC` przerywamy wykonywanie tylko jednej, aktualnej pętli. Polecenie `Stop all computations` przerywa wykonywanie wszystkich poleceń, również tych czekających w kolejce do uruchomienia.

- **Buffered output**

  Ta opcja określa, czy konsola ma być buforowana czy nie. Buforowanie może przyspieszyć odrobinę działanie, ale powoduje, że niektóre wyniki (np. wyświetlone funkcją `cat()`) nie ukazują się natychmiast na ekranie.

- **Word completion**

  Uzupełnianie poleceń. Bardzo przydatna opcja! Wystarczy napisać kilka początkowych liter i nacisnąć klawisz `TAB`, a R uzupełni nazwę funkcji lub obiektu. W sytuacji, gdy jest kilka możliwości uzupełnienia polecenia R, wyświetla na ekranie wszystkie możliwości.

- **Filename completion**

  Uzupełnianie ścieżek do plików. Działa podobnie jak uzupełnianie poleceń. Bardzo wygodne przy wprowadzaniu ścieżek do plików znajdujących się poza aktualnym katalogiem roboczym.

- **List objects**

  Działa jak funkcja `ls()`, czyli wyświetla nazwy wszystkich obiektów w aktualnej przestrzeni roboczej R.

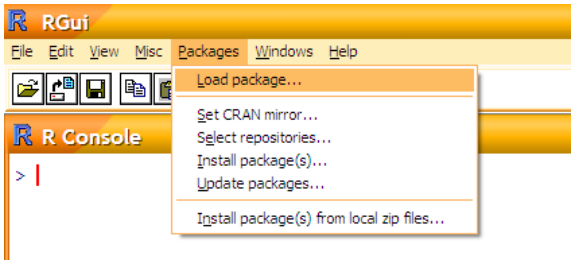
- **Remove all objects**

  Usuwa wszystkie obiekty z przestrzeni roboczej R.

- **List search path**

  Wyświetla nazwy pakietów oraz przestrzeni nazw, w których przeprowadzane będzie wyszukiwanie przez funkcje `search()`.

- **Packages**



  - **Load package...**

    Odpowiednik funkcji `library()`. Powoduje włączenie (załadowanie) wybranego pakietu R.

    Wybierać możemy z listy wszystkich zainstalowanych pakietów.

  - **Set CRAN mirror**

    Umożliwia wybór serwera mirror CRAN, czyli miejsca, z którego ściągane będą pakiety. Domyślne, przy instalacji pierwszego pakietu, R pyta się z jakiego serwera CRAN korzystać i zapamiętuje ten wynik przy kolejnych próbach instalowania pakietów.

Repozytoria to duże zbiory specjalizowanych programów/pakietów/bibliotek.

  - **Select repositories**

    Wskazuje repozytoria, w których wyszukiwane mają być pakiety. Trzy duże repozytoria z pakietami dla R to CRAN (Comprehensive R Archive Network – zbiór serwerów z pakietami i innymi materiałami o R), Omegahat (serwisy poświęcone komunikacji R z innymi językami programowania oraz pakietami do obliczeń statystycznych) oraz Bioconductor (zbiór pakietów dla bioinformatyków, wyspecjalizowanych do analizy danych genomicznych, głównie mikromacierzowych).

- **Install package(s)...**

  Instaluje nowe pakiety. Można wybierać z listy pakietów znajdujących się we wskazanych repozytoriach.

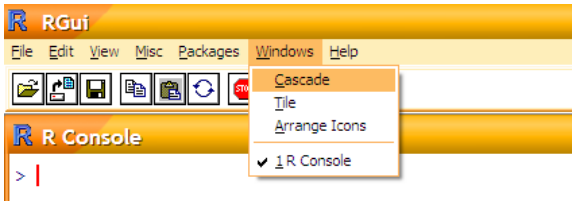
- **Update packages**

  Uaktualnia wskazane pakiety (o ile są dostępne nowsze wersje).

- **Install packages from local zip files**

  Instaluje pakiety z plików zip dostępnych lokalnie. Możemy z serwera CRAN ściągnąć pakiet na przenośny dysk, gdzieś gdzie mamy szybki dostęp do Internetu, a następnie przy pomocy tego polecenia zainstalować na domowym komputerze pakiety ze ściągniętych plików zip.

- **Windows**



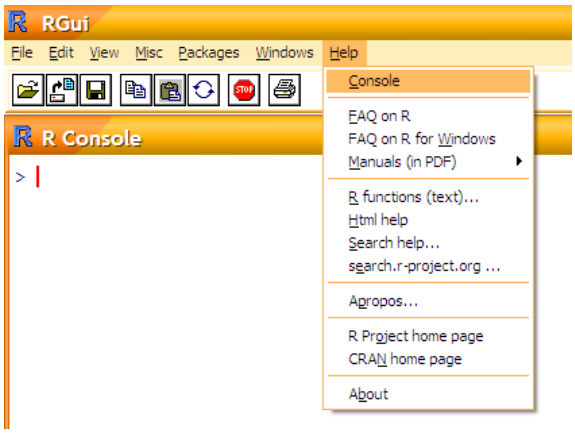
  - **Cascade, Title, Arrange Icons**

    Różne sposoby ułożenia otwartych okien. Przydatne, gdy tych okien jest wiele.

  - **R console, ...**

    Lista otwartych okien. Możemy wybrać, które okno ma zostać uaktywnione.

- **Help**



  - **Console**

    Lista skrótów klawiszowych. Warto ją przeczytać, znajomość tych skrótów może znacznie przyśpieszyć pracę. Przydatne skróty to między innymi: `Ctrl+U` (kasowanie zawartości aktualnej linii); `Ctrl+L` (czyszczenie zawartości okienka konsoli); `Ctrl+T` (zamiana miejscami dwóch sąsiednich znaków); `Ctrl+Tab` (przełączanie do kolejnego otwartego okna R); `ESC` (przerwanie obliczeń).

  - **FAQ on R**

    Lista odpowiedzi na najczęściej zadawane pytania dotyczące R.

  - **FAQ on R for Windows**

    Lista odpowiedzi na najczęściej zadawane pytania dotyczące R w środowisku Windows.

  - **Manuals**

    Zbiór dokumentów w pdf dotyczących wybranych zagadnień wykorzystywania R.

- **R functions (text)**

  Pomoc dotycząca wybranej funkcji.

- **HTML help**

  Strona z pomocą w formacie HTML. Format HTML jest bardzo wygodny do przegladania plików pomocy do pakietów oraz funkcji.

- **Search help**

  Odpowiednik funkcji `help.search()`, czyli wyszukiwania informacji na dany temat w zainstalowanych pakietach.

- **Search.r-project.org**

  Odpowiednik funkcji `Rsitesearch()`, czyli wyszukiwania informacji na dany temat w stronach związanych z platformą R.

- **Apropos**

  Odpowiednik funkcji `apropos()`. Powoduje wyświetlenie listy funkcji zawierających zadany ciąg znaków.

- **R project home page**

  Otwiera w przeglądarce stronę domową R, czyli stronę o adresie `http://www.r-project.org/`.

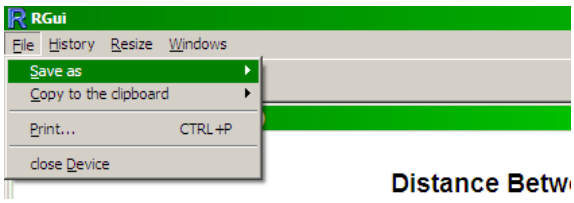
- **CRAN home page**

  Otwiera w przeglądarce wybrane repozytorium CRAN. Adres wrocławskiego repozytorium to `http://r.meteo.uni.wroc.pl/`.

- **About**

  Informacja o uruchomionej wersji R.

Powyższe opcje w menu są dostępne, gdy aktywna jest konsola do wprowadzania poleceń. Jeżeli aktywne jest okno graficzne, to menu jest takie jak przedstawiono poniżej.



- **File**

  - **Save as**

    To polecenie umożliwia zapis rysunku z okna graficznego do pliku w jednym z wielu dostępnych formatów graficznych (pdf, ps, png, bmp, jpg). Rozdzielczość zapisanego rysunku będzie taka, jak wielkość aktualnie otwartego okna graficznego.

  - **Copy to clipboard**

    To polecenie umożliwia skopiowanie rysunku z okna graficznego do schowka Windows. Następnie możemy ten rysunek wkleić do innej aplikacji (np. Word, paint).

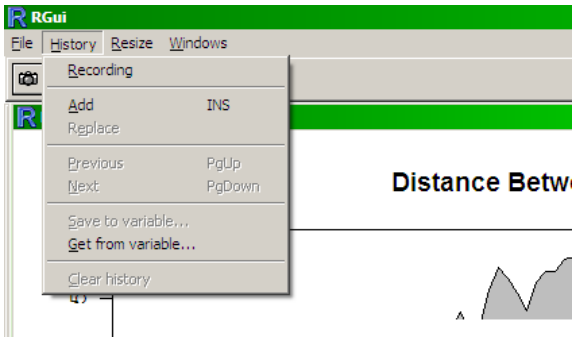
  - **Print**

    To polecenie umożliwia wydrukowanie rysunku.

  - **Close device**

    To polecenie zamyka aktualne okno graficzne.

- **History**



  - **Recording..**

    Włączenie tej opcji powoduje, że R zapamiętuje rysunki, które będą pojawiały się w oknie graficznym. Używając poniżej opisanych poleceń, można przeglądać kolejne rysunki z historii wyświetleń. Przydatna opcja, szczególnie w sytuacji, gdy jakaś funkcja rysuje dwa rysunki, jeden po drugim. Dzięki historii możemy powrócić do poprzedniego rysunku.

  - **Add, Replace, Previouse, Next**

    Te polecenia umożliwiają poruszanie się po historii wyświetleń rysunków w oknie graficznym. Dwóm ostatnim opcjom odpowiadają skróty klawiszowe `PgUp` (przełącz na poprzedni rysunek), `PgDown` (przełącz na następny rysunek).

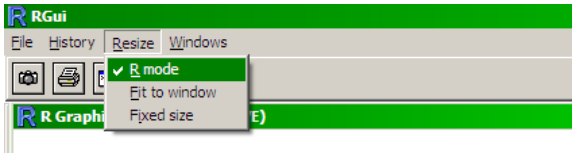
  - **Save to variable, get from variable**

    Umożliwia zapisanie i odtworzenie informacji o oknie graficznym ze zmiennej dostępnej w przestrzeni roboczej R. Przydaje się, jeżeli zawartość okna graficznego ma zostać zmieniona, ale chcielibyśmy zapisać jego aktualny stan.

  - **Clean history**

    Usunięcie informacji o historii wyświetlanych rysunków.

- **Resize**



  Pozwala na określenie wymiarowania rysunku względem rozmiarów okienka graficznego.

- **Windows**

  Identyczne z menu „Windows”, gdy aktywna jest konsola do wprowadzania poleceń.

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)

1.5.3 Gdzie szukać pomocy?

Jest prawdą absolutną, że w przypadku, gdy nie wiemy jak coś zrobić, to najłatwiej i najszybciej będzie 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()` bez argumentów. Wyświetla stronę powitalną systemu pomocy R. Na tej stronie opisane są szczegółowo wymienione poniżej funkcje.
- Funkcja `help("nazwaFunkcji")` lub `?nazwaFunkcji`. Wyświetla stronę z pomocą dla funkcji o nazwie `nazwaFunkcji`. Format opisów funkcji jest ujednolicony, tak aby łatwiej 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 określimy argument `package`, to uzyskamy pomoc dotyczącą konkretnego pakietu. Przykładowo polecenie `help(package=MASS)` wyświetla opis dla pakietu `MASS`.
- Funkcja `args(nazwaFunkcji)`. Wyświetla listę argumentów dla danej funkcji.
- Funkcja `apropos(slowo)` lub `find(slowo)`. Wypisuje listę funkcji (oraz obiektów), które w swojej nazwie mają podciąg `slowo`.
- Funkcja `example(nazwaFunkcji)`. Uruchamia skrypt z przykładowymi wywoływaniami poszczególnych funkcji. 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 pozycje, 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 przykładowa sesja w pakiecie R, poszukujemy dodatkowych informacji o funkcji `plot()` oraz o funkcjach do testowania statystycznego.

```
# wyświetl pomoc dotyczącą funkcji plot()
?plot
# wyświetl przykłady użycia funkcji plot()
example(plot)
# wyświetl nazwy funkcji ze słowem "test" w nazwie
apropos("test")
# wyświetl nazwy funkcji ze zwrotem 'normality test' w opisie
help.search("normality test")
```

Powyżej przedstawione funkcje wyszukują informacje na dany 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:

- Poradniki (manuale, ang. *manuals*) do R'a, poświęcone różnym aspektom programowania w R lub analizie danych w R. Dostępne są bezpośrednio z menu `Help` w R (gdy aktywna jest konsola) oraz w Internecie pod adresem `http://cran.r-project.org/manuals.html`.
- 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`.
- Encyklopedia Rwiki dostępna pod adresem `http://wiki.r-project.org/rwiki/`. Jest to Wiki o R, czyli masa ciekawych informacji w mniej lub bardziej kontrolowanej formie (szczególnie warto przejrzeć sekcje zatytułowaną „R graph galery").
- Wyszukiwarka Rseek dostępna pod adresem `http://www.rseek.org/`. Jest to potężna wyszukiwarka funkcji, obiektów, komentarzy i innych informacji. Jeżeli nie znajdzie się tego czego się szuka, to zawsze można też zapytać wyszukiwarkę google przeglądając tylko podstrony serwisu o R. Tak ustawiona wersja google znajduje się np. pod adresem `http://www.r-project.org/search.html`.
- Lista dyskusyjna poświęcona rozwiązywaniu problemów w korzystaniu z R. Jej archiwum znajduje się pod adresem `http://www.r-project.org/mail.html`. W razie problemów ze znalezieniem odpowiedzi na frapujący nas problem można tu zadać pytanie i cierpliwie czekać na odpowiedź. W 99% przypadków już ktoś zadał takie pytanie i uzyskał odpowiedź, warto wiec najpierw przejrzeć archiwum listy.
- Zbiór porad FAQ dostępne pod adresem `http://www.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 *Frequently Asked Question*).
- Galeria grafik dostępna pod adresem `http://addictedtor.free.fr/graphiques/`. To strona internetowa ze zbiorem różnych ciekawych wykresów wykonanych w R. Warto zerknąć, bo z pewnością robi wrażenie. Co więcej kody R wykorzystane do wykonania poszczególnych wykresów można ściągnąć i samodzielnie przeanalizować.

Powyższe źródła są bez wyjątku angielskojęzyczne. Poza nimi w Internecie można znaleźć też wiele materiałów polskojęzycznych. W szczególności warto przejrzeć dokument *„Wprowadzenie do środowiska R"* autorstwa Łukasza Komsty [13] dostępny pod adresem `http://cran.r-project.org/doc/contrib/Komsta-Wprowadzenie.pdf`. W polskim Internecie można też znaleźć wiele notatek do wykładów lub laboratoriów prowadzonych z użyciem pakietu R.

W razie wątpliwości lub problemów zawsze można zadać pytanie na którymś z polskich forów, na którym pojawiają się użytkownicy pakietu R, np. na forum poświęcone statystyce, znajdujące się pod adresem `http://www.statystycy.pl/` lub na forum dedykowane pakietowi R i jego użytkownikom, które jest dostępne pod adresem `https://www.im.uj.edu.pl/gur/`.

1.5.4 kalkuRator

R to bardzo potężny, zaawansowany i rozbudowany pakiet statystyczny. Ale można 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 komendy. Naciśnięcie klawisza `ENTER` powoduje zakończenie linii i (o ile to możliwe) wykonanie polecenia.

Poniżej przedstawiamy przykładową sesję z pakietem R w roli kalkulatora.

```
> 2+2          # na początek coś postego
[1] 4
> 2^10 -1      # dwie operacje, potęgowanie ma wyższy priorytet
[1] 1023
> 1/5
[1] 0.2
> sin(pi/2)    # funkcje trygonometryczne operują na radianach
[1] 1
> sin(pi/3)^2 + cos(pi/3)^2  # pamiętamy z trygonometrii skąd ten wynik?
[1] 1
> (3+7)^(4-2)
[1] 100
> atan2(1,1)    # wywołanie funkcji arcus tangens, patrz tabela 1.1
[1] 0.7853982
> pi/4
[1] 0.7853982
> log(1024,2)
[1] 10
> choose(6,2)   # symbol Newtona, nie każdy kalkulator potrafi go wyliczyć
[1] 15
```

Napis `[1]` rozpoczynający linię z wynikiem związany jest ze sposobem działania funkcji wyświetlającej liczby. Mianowicie, jeżeli wyświetlane są wartości długiego wektora liczb, to w nawiasie kwadratowym znajduje się indeks elementu wyświetlanego bezpośrednio za tym nawiasem. W prezentowanych przypadkach wynikiem jest jedna liczba, która jest traktowana przez R jako jednoelementowy wektor, stąd napis `[1]`. Jeszcze do tego wrócimy.

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

Jak widać liczenie w R to nic trudnego. Do dyspozycji mamy wszystkie popularne operatory arytmetyczne (ich lista znajduje się w tabeli 1.1). Wyrażenia arytmetyczne można grupować wykorzystując nawiasy **()**. W R dostępne są również popularne funkcje arytmetyczne (ich lista znajduje się w tabeli 1.3), oraz najpopularniejsze funkcje trygonometryczne (wymienione w tabeli 1.2). 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.

Warto zwrócić uwagę na funkcje `expm1(base)` i `log1p(base)`. 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)     # Opis tych funkcji znajduje się w tabeli 1.3
[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 operatorów arytmetycznych

`- x`	Zmiana znaku `x`.
`x + y` (`x - y`)	Suma (różnica) dwóch liczb `x` i `y`.
`x * y` (`x / y`)	Iloczyn (iloraz) dwóch liczb `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 1.2: Lista funkcji trygonometrycznych z pakietu *base*

`cos(x)`	Wartość funkcji cosinus w punkcie `x`.
`sin(x)`	Wartość funkcji sinus w punkcie `x`.
`tan(x)`	Wartość funkcji tangens w punkcie `x`.
`acos(x)`	Wartość funkcji arcus cosinus w punkcie `x`.
`asin(x)`	Wartość funkcji arcus 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 1.3: 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 1.4: Lista funkcji specjalnych i do operacji na liczbach zespolonych (pakiet *base*)

`beta(a,b)`	Wartość funkcji $\mathcal{B}(a,b)$ o argumentach `a` i `b`.
`lbeta(a,b)`	Wartość logarytmu z funkcji $\mathcal{B}(a,b)$.
`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 $\Gamma(x)$.
`combn(n,k)`	Lista wszystkich kombinacji k elementowych ze zbioru n elementowego.
`choose(n,k)`	Liczba kombinacji k elementowych ze zbioru n elementowego.
`lchoose(n,k)`	Logarytm z liczby 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, imaginary=0, modulus=1,argument=0)`	Funkcja do konstruowania liczb zespolonych. Liczby możemy określać podając część rzeczywistą i urojoną lub podając moduł i argument.
`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`.

1.5.5 Kilka przykładowych sesji w R

W dalszej części tej książki na przykładach pokażemy, co można robić w 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 przykładów pracy z R, tak by łatwiej było przedzierać się przez późniejsze, sformalizowane opisy. Aby zdobyć biegłość w programowaniu w 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 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 R. W tym celu należy wpisać zawartość wszystkich linijek rozpoczynających się od znaku > (znaku zachęty > nie przepisujemy, jedynie to co jest za nim).

```
> # zaczynamy od przypisania wartości do zmiennych a i b
> a = 3
> b = 5
> # 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 zmuszamy R do wypisania
    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 2.5.1 poznamy bardziej zaawansowane sposoby radzenia sobie z błędami.

```
> # gdy użyjemy nazwy zmiennej, która nie została zadeklarowana to
    zgłoszony będzie taki błąd, najczęściej oznacza on złe wpisanie nazwy
    zmiennej, literówkę itp.
> 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 (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.

```
> # pierwsza próba, niestety bez powodzenia
> 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 wyraźnie dać do zrozumienia platformie R
> sqrt(-17+0i)
[1] 0+4.123106i
> (2+4i)*(3-2i)
[1] 14+8i
```

A teraz skonstruujemy wektor liczb i wykonamy na nim kilka operacji. Prześledźmy uważnie wyniki poniższych instrukcji.

```
> # wektor tworzy się korzystając z funkcji c()
> (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
> # zamiast wpisywać długie sekwencje liczb ręcznie możemy je generować
    automatycznie
> 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 wykonywaliśmy operacje na całym wektorze. Możemy również manipulować fragmentami 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[1]
[1] 11
> # co jest w elemencie 2 i 3 wektora
> 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 2×3 i wykonamy na niej kilka operacji arytmetycznych.

```
> # tworzymy nową macierz złożoną z samych zer
> macierz = matrix(0,2,3)
> # wyświetlmy ją
> macierz
     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    0
> # tak jak w przypadku wektora na macierzy możemy wykonywać operacja
    arytmetyczne
> macierz+1
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    1    1    1
```

```
> # a teraz tworzymy inną, ciekawszą 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
> det(A)
[1] -2
> # i macierz odwrotną do A
> 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
```

To tyle tytułem rozgrzewki, nadszedł czas na trochę teorii.

1.5.6 Podstawy składni języka R

Poniżej przedstawimy podstawy składni języka R. Przedstawimy 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.

1.5.6.1 Obiekty

Wszystko czym można operować w języku R jest obiektem. Obiekty można podzielić (nie wdając się w formalne szczegóły) na kilka typów (rodzajów):

- **Typ liczbowy.** Ten typ nie wymaga komentarza. Obiekty tego typu przechowują liczby, zarówno całkowite jak i rzeczywiste. Wpisując liczby dozwolona jest notacja naukowa (np. `2.5e3`). Kropką dziesiętną w R jest kropka. Wyróżnioną wartością jest `NaN` (to skrót rozwijający się w „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` określają plus i minus nieskończoność.

```
> 1
[1] 1
> 1.5
[1] 1.5
> 1.5e5
[1] 150000
```

- **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 R wektor danych opisujących płeć, najlepiej użyć typu czynnikowego. Zmienne tego typu są najczęściej wykorzystywane do definiowania grup. Gdy możemy, warto używać tego typu dla poprawienia efektywności. Zmienne typu czynikowego zajmują mniej miejsca w pamięci niż odpowiadające im łańcuchy znaków, można na nich szybciej wykonywać określone funkcje. Ponadto wiele funkcji R (szczególnie statystycznych) jest w stanie rozpoznać, że argument jest typu wyliczeniowego i zastosować odpowiednie działania, np. wyznaczyć liczebności poszczególnych grup itp.

  Konstruktorem tego typu jest funkcja `factor()`. Na poniższym przykładzie konstruujemy wektor elementów typu wyliczeniowego z dwoma poziomami.

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.

```
> (nz = factor(c("sierzant", "kapitan", "sierzant", "sierzant")))
[1] sierzant kapitan  sierzant sierzant
Levels: kapitan sierzant
> # zobaczmy opis tego wektora
> summary(nz)
 kapitan sierzant
       1        3
```

- **Typ znakowy.** Wartościami obiektów tego typu są napisy (będziemy też używać nazwy łańcuchy znaków). W 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 `T` lub `TRUE`) albo logiczny fałsz (oznaczany przez literał `F` lub `FALSE`). 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.

```
> 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
```

- **Wektor elementów.** Wektor to uporządkowany zbiór obiektów tego samego typu. Do tworzenia wektora z pojedynczych elementów lub innych wektorów służy funkcja `c()`. 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.

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)

```
> 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
```

Język R został tak zaprojektowany, by operacje na wektorach były możliwie najefektywniejsze. Nawet pojedyncza wartość jest traktowana jako wektor jednoelementowy.

Wszystkie elementy wektora muszą mieć ten sam typ. Wyjątkiem jest umieszczanie w wektorze dowolnego typu wartości `NA` (*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, gdy jest ustawiony na T pozwala na usuwanie brakujących obserwacji przed kontynuowaniem obliczeń.

```
> wektor = c(1, 2, NA, 40, 51)
> # funkcja mean() liczy średnią arytmetyczną, ale co ma zrobić z
    brakującą obserwacją?
> mean(wektor)
[1] NA
> # dodanie argumentu na.rm=T rozwiązuje problem
> mean(wektor, na.rm=T)
[1] 23.5
```

Jeżeli chcemy usunąć z wektora wartości brakujące, to możemy posłużyć się funkcją `na.omit(stats)` (jej wynikiem jest wektor bez elementów NA) lub funkcją `complete.cases(stats)` (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 `complete.cases()` określi dla każdego wiersza, czy znajdują się w nim brakujące obserwacje. Związana z wartościami brakującymi jest również funkcja `na.fail(stats)` 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. Konstruktorem listy jest funkcja `list()`. Do elementów listy możemy się odwoływać jak do elementów wektora lub korzystając z nazw poszczególnych pól. W poniższym przykładzie konstruujemy listę złożoną 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.** Konstruktorem macierzy dwuwymiarowej jest funkcja `matrix()`. Parametrami tej funkcji jest wektor liczb inicjujących wartości macierzy oraz dwie liczby określające wymiary macierzy. 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
```

Można też konstruować macierze o większej liczbie wymiarów, ten temat poruszymy w kolejnych podrozdziałach.

- **Ramka danych.** Szczególnym typem obiektu jest ramka danych, którą można traktować jak listę wektorów o tej samej długości. Ramka danych może być wyświetlana jako macierz, w której elementy w kolumnie są tego samego typu, ale mogą różnić się typem pomiędzy kolumnami. Konstruktorem ramki danych jest funkcja `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(T,T,F)))
   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[,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.

1.5.6.2 Konwersja

Typ zmiennej nie jest przypisany do zmiennej (czy jej wartości) na stałe. Możemy zmieniać typy nie podając nowej wartości dla zmiennej. Proces zmiany typu nazywamy konwersją typu.

Najczęstsze konwersje to zamiana na typ znakowy (funkcja `as.character(base)`) lub na typ liczbowy (funkcja `as.numeric(base)`). 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 funkcji konwertujących typ zmiennej jest przedstawiona w tabeli 1.5.

Tabela 1.5: Funkcje pozwalające na sprawdzenie lub konwersję typu zmiennej

`is.numeric(base)`	Test czy argument jest liczbą.
`is.integer(base)`	Test czy argument jest liczbą całkowitą.
`is.double(base)`	Test czy argument jest liczbą rzeczywistą.
`is.complex(base)`	Test czy argument jest liczbą zespoloną.
`is.logical(base)`	Test czy argument jest wartością logiczną.
`is.character(base)`	Test czy argument jest znakiem lub łańcuchem znaków.
`is.factor(base)`	Test czy argument jest typu wyliczeniowego.
`is.na(base)`	Test czy argument jest nieokreślona (`NA`).
`is.nan(base)`	Test czy argument jest niewłaściwą liczbą (`NaN`).
`as.numeric(base)`	Konwersja na wartość liczbową.
`as.integer(base)`	Konwersja na wartość całkowitoliczbową. Liczby rzeczywiste są zaokrąglane w dół.
`as.double(base)`	Konwersja na wartość rzeczywistą.
`as.complex(base)`	Konwersja do liczby zespolonej.
`as.logical(base)`	Konwersja na wartość logiczną.
`as.character(base)`	Konwersja na typ znakowy.
`as.factor(base)`	Konwersja na typ wyliczeniowy.
`as.list(base)`	Konwersja do listy.
`unlist(base)`	Konwersja z listy do wektora.
`as.matrix(base)`	Konwersja na macierz.
`as.data.frame(base)`	Konwersja na ramkę danych.

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.

```
> (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, tak jak pozostałe funkcje z tej tabeli.

```
> wektor = 1:6
> # klasa zmiennej wektor zwraca klasę elementów tego wektora
> 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, o czym jeszcze powiemy więcej w kolejnych rozdziałach. Funkcja *mode()* informuje o wewnętrznej reprezentacji danej zmiennej i nie możemy na wynik tej funkcji bezpośrednio wpływać. Więcej o tych funkcjach przeczytać można w podrozdziale 2.1.6.

1.5.6.3 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 jednym z trzech operatorów przypisania (można też inaczej, ale o tym w uwadze na koniec tego podrozdziału):

- operator `=` przypisuje wartość znajdującą się z prawej strony do zmiennej znajdującej się po lewej stronie,
- operator `<-` jak wyżej,
- operator `->` przypisuje wartość znajdującą się z lewej strony do zmiennej znajdującej się po prawej stronie.

Do zmiennej wartość można również przypisać korzystając z funkcji `assign(base)`, ale korzystanie z powyższych operatorów jest wygodniejsze. Poniżej różne przykłady przypisania wartości.

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

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 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.

```
> a = b = 5     # wszystko ok, obie zmienne mają wartość 5
> a <- b <- 5   # wszystko ok, obie zmienne mają wartość 5
> a = b <- 5    # wszystko ok, obie zmienne mają wartość 5
> a <- b = 5    # mamy problem, lewa strona operatora = nie jest zmienną
Error in (a <- b) = 5 : could not find function "<-<-"
```

1.5.6.4 Indeksy

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

- 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
> wektor <- c(a=1, b=2, c=3)
> # indeksy nie muszą być liczbami, mogą być nazwami elementów
> wektor[c("a","c")]
a c
1 3
```

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

  Gdzie `zmienna` jest macierzą lub ramką danych. Wybierana jest podmacierz (ramka danych) o wskazanych indeksach. Jeżeli któryś z zakresów nie będzie podany zostaną wybrane wszystkie elementy w danym wierszu/kolumnie.

```
> macierz <- matrix(1:4,2,2)
> macierz
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> macierz[1,]      # tylko pierwszy wiersz
[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 nie jest już macierzą ale wektorem. Jeżeli chcemy zagwarantować, że wynik będzie macierzą, to należy zmienić argument `drop`. Poniżej przykład.

```
> macierz[1,]           # wynik jest wektorem
[1] 1 3
> macierz[1,,drop=T]    # wynik jest wektorem
[1] 1 3
> macierz[1,,drop=F]    # wynik jest macierzą
     [,1] [,2]
[1,]    1    3
```

- W notacji listowej *zmienna$wartosc1*

  Gdzie `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(name=c("Jan","Tomasz"), surname="Kowalski", age=25,
    married=T)
> osobnik$name
[1] "Jan"    "Tomasz"
> osobnik[[2]]
[1] "Kowalski"
```

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

  Gdzie `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 do 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)
> # drugi element macierzy to jedna liczba
> macierz[[2]]
[1] 2
> osobnik <- list(name=c("Jan","Tomasz"), surname="Kowalski", age=25,
    married=T)
> # pierwszy element listy, wektor dwuelementowy
> osobnik[[1]]
[1] "Jan"    "Tomasz"
```

Przydatną funkcją przy operacjach na indeksach jest funkcja `which(base)`. Jej wynikiem są indeksy elementów spełniające zadany warunek logiczny. Jeżeli interesują nas indeksy wystąpień pewnego wektora elementów w innym wektorze możemy

użyć funkcji `match(base)`. 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(base)`, `which.max(base)`. 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))  # które najmniejsze
[1] 4 6
> which.max(wektor)             # który największy
[1] 2
> which.min(wektor)             # który najmniejszy
[1] 4
> # zmienna LETTERS to 26-elementowy wektor dużych liter
> 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
```

1.5.6.5 Operatory

W tabeli 1.6 przedstawiono najpopularniejsze operatory w języku R. Dla większości operatorów istnieją również funkcje o identycznym działaniu, ale mniej wygodnym zapisie. Np. wyrażenie `2+3` jest równoważne wyrażeniu `sum(2,3)`. Można również definiować własne operatory, co opiszemy w podrozdziale 1.6.2.5.

Tabela 1.6: Lista operatorów logicznych i arytmetycznych

`+`, `-` , `*`, `/`, `^`	Standardowe operatory arytmetyczne. Oba argumenty powinny mieć taki sam wymiar (macierze lub wektory) lub jeden z argumentów powinien być liczbą.
`%x%`	Iloczyn Kroneckera dwóch macierzy.
`%%`	Reszta modulo z dzielenia.
`%/%`	Dzielenie całkowite.
`%*%`	Iloczyn dwóch macierzy.
`<`, `==`, `>`, `<=`, `>=`, `!=`	Standardowe operatory porównywania wartości liczbowych.
`!`	Operator negacji.
`&`, `&&`, `|`, `||`	Logiczny iloczyn oraz logiczna suma.
`any()`, `all()`	Logiczna suma(iloczyn) wszystkich elementów wektora.

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 (preferujemy bezstresowy sposób nauczania ; >).

Operatory `&` i `|` służą do wykonywania operacji na listach lub wektorach, podczas gdy `&&` i `||` na pojedynczych wartościach. Warto przeanalizować co się dzieje w poniższym przykładzie.

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

```
> # podajemy dwa wektory wartości logicznych, kolejne wartości są nazwane
> lubie.statystyke   = c(ala=FALSE, ola=TRUE, ewa=TRUE)
> lubie.prowadzacego = c(ala=TRUE, ola=TRUE, ewa=FALSE)
> # logiczne i dla kolejnych par elementów obu wektorów
> lubie.statystyke & lubie.prowadzacego
  ala   ola   ewa
FALSE  TRUE FALSE
> # logiczne i dla wszystkich elementów obu wektorów
> lubie.statystyke && lubie.prowadzacego
[1] FALSE
> # logiczne lub dla kolejnych par elementów obu wektorów
> lubie.statystyke | lubie.prowadzacego
 ala  ola  ewa
TRUE TRUE TRUE
> # 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 `&` stosują tak zwane „gorliwe wartościowanie", czyli wyznaczają wartości wszystkich argumentów tych operatorów a następnie wyznaczają logiczną sumę lub iloczyn. Operatory `||` i `&&` stosują 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. Leniwe i gorliwe wartościowanie to przydatny mechanizm ale może być też źródłem błędów.

```
> wypiszImie <- function(imie) {cat(paste(imie, "\n")); T }
> wypiszImie("ala") | wypiszImie("ola")
ala
ola
[1] TRUE
> wypiszImie("ala") || wypiszImie("ola")
ala
[1] TRUE
```

1.5.6.6 Sekwencje liczb

Sekwencje to regularne wektory liczb całkowitych. Takie wektory można generować używając operatora : lub funkcji `seq(base)`. Kilka sposobów generowania sekwencji przedstawimy poniżej.

```
> # z użyciem operatora :, generowana jest sekwencja liczb od ... od ...
      z krokiem 1
> -2:2
[1] -2 -1  0  1  2
> 2:-2
[1] 2 1  0  -1  -2
> # poniższa 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 pożądany wymiar 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(base)`. 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=T`) lub bez zwracania (gdy argument `replace=F`, 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 romańskiego (inne ciekawe predefiniowane wektory to `LETTERS` – duże litery, `month.name` – nazwy miesięcy i `month.abb` – trzyliterowe skróty nazw miesięcy).

```
> # wylosujmy dziesięć losowych liter
> sample(letters,10,T)
 [1] "u" "q" "x" "s" "q" "f" "c" "f" "l" "x"
> # wylosujmy wektor cyfr od 1 do 3, z zadanymi prawdopodobieństwami
    wylosowania
> sample(1:3,20,T, 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
```

1.5.6.7 Komentarze

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

1.5.7 Wyświetlanie i formatowanie obiektów

Dwie najpopularniejsze funkcje do wyświetlania wartości obiektów to: `cat(base)` i `print(base)`. Funkcje te różnią się w działaniu. Aby je porównać zacznijmy od przykładu, w którym wyświetlimy wektor 6 napisów.

```
> nap = rep(c("Ala ma kota", "Ola nie ma kota", "Ela chce mieć kota"),2)
> print(nap)            # wyświetlanie sformatowane
[1] "Ala ma kota"        "Ola nie ma kota"    "Ela chce mieć kota"
[4] "Ala ma kota"        "Ola nie ma kota"    "Ela chce mieć kota"
> cat(nap)                # wyświetlanie niesformatowane
Ala ma kota Ola nie ma kota Ela chce mieć kota Ala ma kota Ola nie ma 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.

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), funkcji `cat()` przeciążać nie można.

Domyślnie, jeżeli w wyniku wykonania polecenia w 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 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()` (przeciążony odpowiednik funkcji `print()`) odpowiedzialna za wyświetlenie podsumowania. 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(base)`. Działanie tej funkcji polega na tymczasowemu zapobiegnięciu wyświetlania jej argumentu w sytuacji gdy nie będzie on do niczego przypisany. Prześledźmy poniższy przykład.

```
> # obie funkcje są tożsamościami
> I1 <- function(x) x
> I2 <- function(x) invisible(x)

> # wynik funkcji przypisujemy do zmiennej,  w tym przypadku funkcja
    invisible() nie ma żadnego efektu, dla obu funkcji obserwujemy to samo
     zachowanie
> a <- I1(1)
> a
[1] 1
> a <- I2(1)
> a
[1] 1
> # wyniku funkcji nie przypisujemy do zmiennej,  w tym przypadku funkcja
    invisible() powoduje, ze wynik 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 1.6.5.1.

Do operacji na napisach w celu ich odpowiedniego wyświetlenia wykorzystuje się również funkcje `paste(base)` i `format(base)`. Funkcja `paste()` służy do łączenia wektorów napisów, jej dwa argumenty `sep=" "` i `collapse=NULL` określają sposób w jaki sklejane są argumenty tej funkcji. Poniżej kilka przykładów.

```
> # Funkcja paste() skleja 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 tej funkcji.

```
> 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. bardzo przydatna jest funkcja `sprintf(base)`, która ma podobny sposób formatowania do funkcji o tej samej nazwie w języku C. Informacji o innych funkcjach do formatowania można szukać np. w plikach pomocy do funkcji `toString(base)` i `encodeString(base)`. Przedstawimy tutaj jeszcze tylko jeden przykład, dotyczący funkcji `formatFix(cwhstring)`. Konwertuje ona wektor liczb (pierwszy argument) na napisy o formacie określonym przez pozostałe argumenty.

```
> require(cwhstring)
> # argument after określa liczbę miejsc po kropce a before przed kropką,
    w wynikowym wektorze wszystkie napisy mają taką samą długość
> formatFix(c(pi, exp(1), 1, 1/pi),after=3, before=3)
[1] "  3.142" "  2.718" "  1.000" "  0.318"
```

Funkcja `require(base)` ładuje bibliotekę, podobnie jak funkcja `library(base)`. 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 Przyśpieszamy

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)

Po lekturze tego rozdziału czytelnik będzie już całkowicie gotów do korzystania z 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.

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.

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` jest następująca:

```
if (war)
    instr1

# lub

if (war)
    instr1  else  instr2
```

Fragment instrukcji warunkowej począwszy od słowa `else` jest nieobowiązkowy. Jeżeli wartość warunku logicznego `war` jest prawdziwa (logiczna równa `TRUE` lub liczbowa różna od 0), to wykonana zostanie `instr1`, jeżeli nie, to wykonana będzie `instr2` (o ile podano wariant z `else`). Zarówno `instr1` jak i `instr2` mogą być zastąpione blokiem instrukcji (instrukcjami otoczonymi nawiasami `{}`).

Zobaczmy, jak to wygląda na poniższym przykładzie.

```
> liczba = 1313
> # zapis 'liczba %% 2 == 0' oznacza sprawdzenie, czy reszta z dzielenia
    prez 2 ma wartość 0, jezeli tak jest, to to wyrażenie przyjmuje
    wartość TRUE, w przeciwnym razie wartość FALSE
> 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:

```
# ta wersja nie będzie działała
if (1==0)
    cat("to nie może być prawda")
  else
    cat("co się wyświetli?")
```

Ale uwaga! Jeżeli te instrukcje znalazły by 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 linijki o instrukcji `if` już nie pamięta, dlatego zostanie zgłoszony błąd składni. Poprawne użycie instrukcji `if` z wariantem `else` jest następujące:

Interpreter to program interpretujący instrukcje zapisane w języku programowania na rozkazy rozumiane przez procesor. Interpreter języka R jest integralną częścią platformy R.

```
# ta wersja będzie działała
if (1==0) {
           cat("to nie może być prawda")
        } else {
          cat("co się wyświetli?")
        }
# podobnie jak ta
if (1==0)
          cat("to nie może być prawda") else
          cat("co się wyświetli?")
```

1.6.1.2 Funkcja ifelse(base)

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. Składnia tej funkcji jest następująca:

```
ifelse(war, instr1, instr2)
```

Warunek `war` może być jedną wartością logiczną lub wektorem wartości logicznych. W wyniku wykonania tej funkcji zwrócona zostanie wartość lub wektor. Wynik będzie miał wartości opisane przez `instr1` w pozycjach odpowiadających wartości `TRUE` wektora `war` oraz wartości opisane przez `instr2` w pozycjach odpowiadających wartości `FALSE` wektora `war`. Prześledźmy wyniki poniższych przykładów.

```
> # pierwszy argument jest wektorem
> ifelse(1:8 < 5, "mniej", "wiecej")
[1] "mniej" "mniej" "mniej" "mniej" "wiecej" "wiecej" "wiecej" "wiecej"
> # wszystkie  argumenty są wektorami
> ifelse(sin(1:5)>0, (1:5)^2, (1:5)^3)
[1]   1   4   9  64 125
> # wszystkie  argumenty to pojedyncze wartości
> ifelse(1==2, " cos jest nie tak", "uff")
[1] "uff"
```

1.6.1.3 Funkcja switch(base)

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.

```
> dane = 1313
> # w poniższej instrukcji sprawdzana jest klasa zmiennej dane, ta klasa
    to 'numeric', ponieważ zmienna 'dane' przechowuje liczbę, dlatego też
    wykona się wyłącznie druga akcja
> switch(class(dane),
+   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 2.1.9). Zwykłe 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 (na szczęście) nie dzieje, ponieważ w tym przypadku 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`. W języku R ta pętla również jest dostępna ale korzysta się inaczej niż w większości języków programowania. Poniżej przedstawiamy składnię pętli `for`.

```
for (zmien in wekt)
                  instr
```

Jeżeli chcemy w każdym wykonaniu pętli wykonać więcej poleceń, to `instr` można zastąpić blokiem instrukcji. Ta instrukcja lub blok instrukcji będzie wykonana tyle razy ile elementów ma wektor `wekt`. Zmienna `zmien` w każdym okrążeniu pętli przyjmować będzie kolejną wartość z tego wektora. Zacznijmy od przykładu.

```
> # ta pętla wykona się dla każdego elementu wektora 1:4
> for (i in 1:4) {
+    cat(paste("aktualna wartosc i to ", i, "\n"))
+ }
aktualna wartosc i to 1
aktualna wartosc i to 2
aktualna wartosc i to 3
aktualna wartosc i to 4
```

Elementy wektora `wekt` mogą być dowolnego typu. Poniżej umieszczamy przykład, w którym indeks funkcji przebiega po wartościach wektora napisów.

Co więcej, *wekt* nie musi być wektorem, może być listą, której elementy są różnego typu!

```
> indeksy = c("jablka","gruszki","truskawki","pomarancze")
> # ta pętla wykona się dla każdego elementu wektora 'indeksy'
> for (i in indeksy) {
+    cat(paste(i, "\n"))
+ }
jablka
gruszki
truskawki
pomarancze
```

Pętla `for` nie wymaga by elementy `wek` były różne ale warto o to zadbać. Ułatwia to śledzenie ewentualnych błędów, wystarczy bowiem wypisać, w którym kroku pętli wystąpił błąd. W większości przypadków wygodnie jest za `wekt` podać wektor kolejnych liczb całkowitych. Umożliwia to zapisywanie wyników z kolejnych okrążeń pętli w wektorze lub w macierzy.

W wielu sytuacjach do wyznaczania wektora indeksów pętli wygodnie jest się posłużyć funkcją `seq_along(base)`. 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.

```
> # ta pętla wykona się dla każdego elementu wektora 'indeksy' ale zmienna
    'i' nie będzie przyjmowała za wartości elementów tego wektora ale
   jego indeksy. Innymi słowy w pierwszym wywołaniu wartością 'i' nie
   będzie wartość 'jablka' ale wartość '1'
>  for (i in seq_along(indeksy)) {
+    cat(paste(indeksy[i], "\n"))
jablka
gruszki
truskawki
pomarancze
```

Zazwyczaj podobny efekt do pętli `for` możemy uzyskać stosując funkcje z rodziny `*apply()` (np. `lapply()`). Funkcje te będą przedstawione w podrozdziale 2.1.3. 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.

1.6.1.5 Pętla while

Pętla `for` ma z góry określoną liczbę powtórzeń do wykonania (wyznaczoną przez długość wektora `wekt`). W pewnych sytuacjach nie wiemy ile powtórzeń pętli będzie wymaganych, aby uzyskać zamierzony efekt. W takich sytuacjach wygodniej jest wykorzystać pętlę `while`. Poniżej przedstawiamy jej składnię.

```
while (war) instr
```

Instrukcja `instr` będzie wykonywana tak długo, dopóki warunek `war` jest prawdziwy. Oczywiście, należy zadbać o to, by taka sytuacja kiedykolwiek zaistniała a więc by pętla się zakończyła. Poniżej przykład z wykorzystaniem pętli `while`.

```
> i=0
> # pętla będzie się wykonywać póki warunek 'i<3' będzie prawdziwy
> while(i < 3) {
+   cat(paste("juz",i,"\n"))
+   i = i+1
+ }
juz 0
juz 1
juz 2
```

1.6.1.6 Pętla repeat

W języku R istnieje też trzeci rodzaj pętli. Przyznam, że niechętnie o nim piszę i raczej bym go nie polecał. Tą czarną owcą jest pętla `repeat`, której składnia przedstawiona jest poniżej.

```
repeat instr
```

Działanie tej pętli polega na powtarzaniu instrukcji `instr` 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`). Poniżej przykład z użyciem tej czarnej owcy.

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

```
> # pętla repeat nie ma określonego warunku stopu, przerwać ją może
    wywołanie polecenia break lub interwencja użytkownika
> 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 w dużych programach pewne fragmenty kodu powtarzają się wielokrotnie i/lub są używane w różnych miejscach naszego programu. Czasem należy wykonać pewien schemat instrukcji, być może z niewielką różnicą w argumentach. W takich sytuacjach wygodnie jest napisać funkcję, która uprości program i poprawi jego czytelność. Generalnie rzecz biorąc zamykanie kodu w funkcjach jest dobrym zwyczajem, umożliwia tzw. reużywalność kodu a tym samym wpływa na zmniejszenie liczby błędów, które zawsze gdzieś w kodzie się znajdą. Jest wiele wskazówek jak pisać dobre funkcje, ponieważ jednak nie jest to podręcznik do informatyki 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). Funkcje powinny być możliwie krótkie, ułatwi to ich modyfikacje i śledzenie. Jeżeli jakaś funkcja znacznie się rozrosła, to z pewnością można i warto podzielić ją na mniejsze funkcje. Poniżej przedstawiamy schemat deklaracji funkcji.

```
function(listaArgumentow)
instr
```

Instrukcje `instr` można zastąpić blokiem instrukcji. Lista argumentów funkcji może być pusta, poniżej zamieszamy przykład deklaracji i wykonania funkcji bez argumentów. Funkcje w języku R są traktowane jak zwykłe obiekty. Konsekwencje takiego rozwiązania zostaną szczegółowo przedstawione później. Poniżej przykład.

```
> # określamy funkcję i przypisujemy ją do zmiennej funkcja1
> funkcja1 = function() {
+   cat("Dzisiaj jest ")
+   cat(format(Sys.time(), "%A %B %d"))
+ }
> # zobaczmy jak działa ta funkcja
> funkcja1()
Dzisiaj jest poniedziałek listopad 05
> # przypisujemy do zmiennej funkcja2 wartość zmiennej funkcja1, a więc
    naszą funkcję
> funkcja2 = funkcja1
> # możemy ją wywołać tak samo, jak funkcję funkcja1
> funkcja2()
Dzisiaj jest poniedziałek 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(utils)` lub `fix(utils)`. Poniższy przykład spowoduje otwarcie okienka edycyjnego, umożliwiający edycję ciała funkcji.

```
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 `listaArgumentow` rozdzielając je 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()`. Prześledźmy poniższy przykład.

```
> # definiujemy nową funkcje, wykorzystaną tutaj funkcję sort opiszemy w
    innym miejscu
> wyswietl3Najmniejsze <- function(wektor) {
+   posortowane <- sort(wektor)
+   posortowane[1:3]
+}
> lLiczby <- c(11, 3, 10, 1, 0, 8)
> # wywołujemy naszą funkcję
> (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(base)`. Przykład mechanizmu skrótów przedstawiony jest poniżej.

```
> # deklarujemy funkcje z dwoma argumentami
> funkcja <- function(liczba = 5, poziom = "sredni")
+             cat(paste(liczba, poziom,"\n"))
> funkcja(3, "duzy")         # wywołanie z jawnym wskazaniem obu arg.
3 duzy
> funkcja(po="duzy")         # wskazujemy drugi argument skrótem nazwy
5 duzy
> funkcja(p="maly", li=1313) # wskazujemy skrótami oba argumenty
1313 maly
```

1.6.2.1 Argumenty domyślne

Definiując funkcje możemy określić domyślne wartości dla kolejnych argumentów funkcji. 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, których modyfikować nie chcemy a w liście argumentów zamiast wartości zostawić puste miejsce.

Wszystkie te możliwości zostały przedstawione w poniższym przykładzie.

```
> # deklarujemy funkcje z trzema argumentami
> wyswietlNajmniejsze <- function(wektor, do = 3, od = 1) {
+   posortowane <- sort(wektor)
+   posortowane[od:do]
+ }
> # możemy wywołać tę funkcję z dowolną kombinacją i kolejnością
    argumentów, poniżej określamy tylko pierwszy argument, pozostałe będą
    domyślne
> 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()`.

```
> # nadmiarowe argumenty zostaną przekazane do funkcji plot()
> narysujNajmniejsze <- function(wektor, ile = 3, ...) {
+   posortowane <- sort(wektor)
+   plot(posortowane[1:ile], ...)
+ }
> # wywołujemy funkcję narysujNajmniejsze() z dodatkowymi argumentami (lwd
    type i col) nie wymienionymi jawnie w liście argumentów
> narysujNajmniejsze(lLiczby, ile=20, lwd=3, type="l", col="black")
```

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

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.

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. Można to zrobić korzystając z funkcji `options(base)`, 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. Jedną z możliwości jest zdefiniowanie takiej funkcji wcześniej, przypisanie jej do jakiejś 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. Wykorzystanie funkcji anonimowych obrazuje poniższy przykład. Korzystamy tu z funkcji `sapply()`, która wykonuje zadaną funkcję (drugi argument funkcji `sapply()`) dla każdego elementu wektora lub listy (pierwszy argument funkcji `sapply()`).

```
> # przykładowe wywołanie funkcji 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 Polimorficzność funkcji

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

Kolejnym mechanizmem zwiększającym możliwości R jest możliwość definiowania funkcji polimorficznych. Mechanizm ten, nazywany też przeciążaniem funkcji, pozwala na imitowanie pewnych cech języka obiektowego.

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 2.1.7). W tym celu należy zdefiniować nową funkcje i przypisać ją do zmiennej o nazwie `plot.typ`, gdzie `typ` to nazwa pewnego typu (klasy, np. `factor`). Wywołanie funkcji o nazwie bez suffiksu `.typ` ale z argumentem klasy `typ` spowoduje wywołanie funkcji z suffiksem `.typ`.

Przykładowo, funkcja `plot()` zazwyczaj rysuje coś na ekranie. Możemy jednak zażyczyć sobie aby w zależności od klasy 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.

```
> # deklarujemy funkcję o nazwie 'plot.logical'
> plot.logical <- function( obj) {
+  cat(ifelse(obj,"prawda","nieprawda"))
+ }
> # funkcja 'plot.logical' będzie wywołana, jeżeli za nazwę funkcji podamy
    'plot' a argument będzie typu 'logical'
> 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()`.

```
> # lista przeciążeń 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 R wiedział, że jakaś funkcja jest generyczna (czyli może być przeciążana) należy to określić 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.

```
> # wskazujemy, że funkcja rozmiar będzie przeciążana
> rozmiar = function(x) UseMethod("rozmiar")
> # opisujemy jej domyślne zachowanie
> rozmiar.default    = function(x) length(x)
> # specyfikujemy jej zachowanie dla konkretnych typów argumentów
> rozmiar.character  = function(x) length(x)
> rozmiar.matrix     = function(x) dim(x)[1] * dim(x)[2]
> rozmiar.array      = function(x) prod(dim(x))
>
> # krótki test, wywołujemy tę funkcję dla wektora
> 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 zakresem funkcji. Warto przeanalizować poniższy przykład.

```
> # definiujemy nową funkcję, w której będziemy przypisywać wartości
> przyklad1 <- function() {
+   # pierwsze przypisanie to normalne, lokalne przypisanie
+       zmienna1 <- 2
+   # drugie przypisanie to globalne przypisanie, modyfikowana jest
    zmienna w zewnętrznej przestrzeni nazw
+       zmienna2 <<- 2
+       cat(paste("zmienna1:",zmienna1,"zmienna2:",zmienna2,"\n"))
+ }
> # zainicjujmy wartość dwóch zmiennych
> zmienna1 = 1
> zmienna2 = 1
> # 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 Własne operatory

Kolejnym rozszerzeniem języka R, jest możliwość definiowania własnych operatorów. Operatorem w R 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%.

```
> # definiujemy operator tak jak zwykłą funkcję dwuargumentową
> "%v%" <- function(x,y)
+    x*y+x+y
> # używamy w sposób typowy dla operatorów
> 1 %v% 2
[1] 5
```

1.6.2.6 Inne zagadnienia związane z funkcjami

Ponieważ funkcje to zwykłe obiekty zatem nic nie stoi na przeszkodzie w definiowaniu funkcji zagnieżdżonych.

```
> # przykład dla zagnieżdżonych funkcji
> zewnetrzna <- function(x) {
+   wewnetrzna <- function(x) {
+      print(x)
+   }
+   wewnetrzna(x)
> }
```

Możemy również funkcję zapisać do pliku, np. funkcją `save(base)`!

W R jest bardzo wiele użytecznych funkcji. 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(base)` i `.Last(base)`. 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 R ma robić po uruchomieniu środowiska?
> .First <- function()
> options(prompt="# ", continue="- \t")
> # co R ma robić przed zamknięciem środowiska?
> .Last <- function()
> cat("baj baj")
```

Jeżeli wprowadzimy te funkcje a następnie zapiszemy całe środowisko w domyślnym *workspace* (jest on ładowany przy każdym uruchomieniu R), to przy kolejnym uruchomieniu 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.

1.6.3 Zarządzanie obiektami w przestrzeni nazw

Przestrzeń nazw to zbiór nazw wszystkich zmiennych, które zostały zadeklarowane na tym samym poziomie. W głównej przestrzeni nazw znajdują się nazwy zmiennych, które określiliśmy globalnie. Zmienne określone 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. Innym argumentem funkcji `ls()` jest `all.names`. Domyślnie argument `all.names` jest ustawiony na `FALSE`, co powoduje, że nazwy obiektów rozpoczynające się od znaku `.` nie są wyświetlane.

Podobne działanie do `ls()` ma funkcja `objects(base)` oraz `ls.str(base)`. Funkcja `ls.str()` również wyświetla listę nazw zmiennych w aktualnej przestrzeni nazw oraz dodatkowo wyświetla informacje o strukturze poszczególnych zmiennych (strukturę jednego obiektu opisuje funkcja `str()`). Aby uzyskać listę funkcji w przestrzeni nazw można użyć funkcji `lsf.str(utils)`.

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. Aby usunąć wszystkie zmienne z przestrzeni roboczej należy użyć polecenia `rm(list = ls())`. 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 przez R uruchamiany automatycznie gdy tylko zachodzi taka potrzeba. Można również ten proces uruchomić ręcznie wywołując funkcję `gc(base)`.

Funkcją `save.image(base)` 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 współpracownikowi. Wybraną zmienną lub zmienne możemy zapisać poleceniem `save(base)`. Inną funkcją z tej serii jest `savehistory(bases)` zapisująca historię wszystkich wykonanych poleceń do wskazanego pliku. Dodatkowe informacje o stanie środowiska R, np. lista załadowanych aktualnie pakietów, można uzyskać uruchamiając funkcję `sessionInfo(utils)`.

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(base)`. Usuwa ona niedozwolone znaki z łańcucha znaków będącego jej argumentem.

Prawdę mówiąc w 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()`. Oczywiście nie polecam tworzenia takich nazw ale przykład ten może się przydać w sytuacjach gdy R wygeneruje automatycznie jakąś bardzo dziwną nazwę dla zmiennej, np. nazwą będzie ścieżka do jakiegoś pliku.

```
> # zaczynamy od dziwnego przypisania, o sposobach przypisywania jeszcze
    będziemy pisać
> assign("a\a", 3)
> # sprawdźmy czy ta zmienna jest w pamięci R
> ls()
[1] "a\a"
> # odczytajmy jej wartość
> get("a\a")
[1] 3
```

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

Michael Rennie fortune(68)

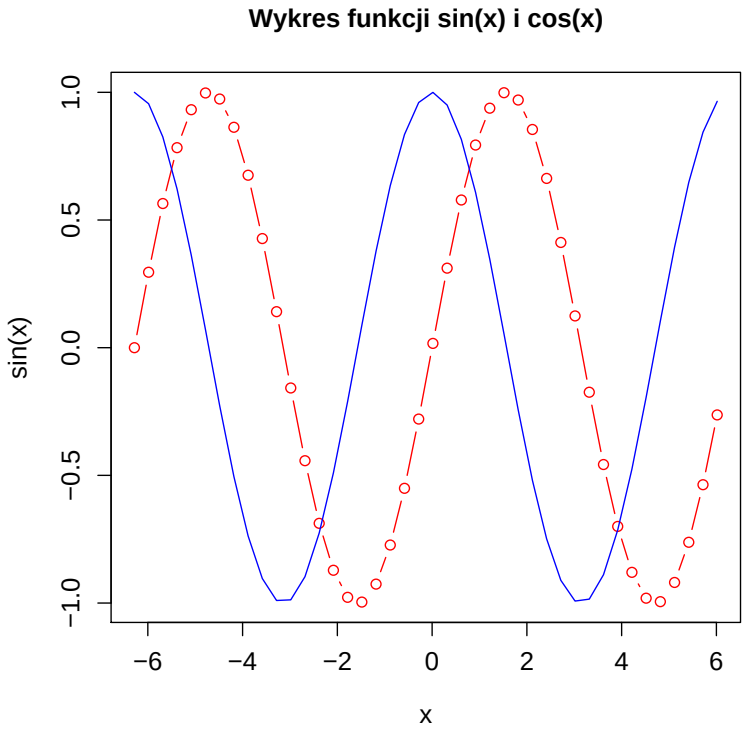
1.6.4 Wprowadzenie do grafiki

Pisaliśmy już wiele o tym, że możliwości graficzne R są olbrzymie. Czas na poznanie ich bliżej. Wykresy w R tworzyć można korzystając z wielu różnych funkcji ale najpopularniejszym sposobem jest skorzystanie z funkcji `plot()` (jest to też jedna z najczęściej przeciążanych funkcji, co oznacza, że ma bardzo wiele wyspecjalizowanych implementacji). Poznawanie grafiki zacznijmy od prostego przykładu.

```
# przygotowujemy siatkę punktów
x = seq(-2*pi,2*pi,by=0.3)
# rysujemy funkcję sin(x)
plot(x, sin(x), type="b",main="Wykres funkcji sin(x) i cos(x)",col="red")
# a następnie dorysowujemy do niej funkcje cos(x)
lines(x, cos(x), col="blue", type="l")
```

W pierwszej linii powyższego przykładu tworzony jest wektor liczb z użyciem funkcji `seq()`. Druga linijka, to wywołanie funkcji `plot()`. Funkcja ta czyści okno graficzne i przygotowuje je do narysowania wykresu. Inicjuje osie, ustawia układ współrzędnych, określa rozmiary marginesów i robi wiele innych przygotowawczych 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).



Rysunek 1.5: Wykres narysowany różnymi rodzajami linii

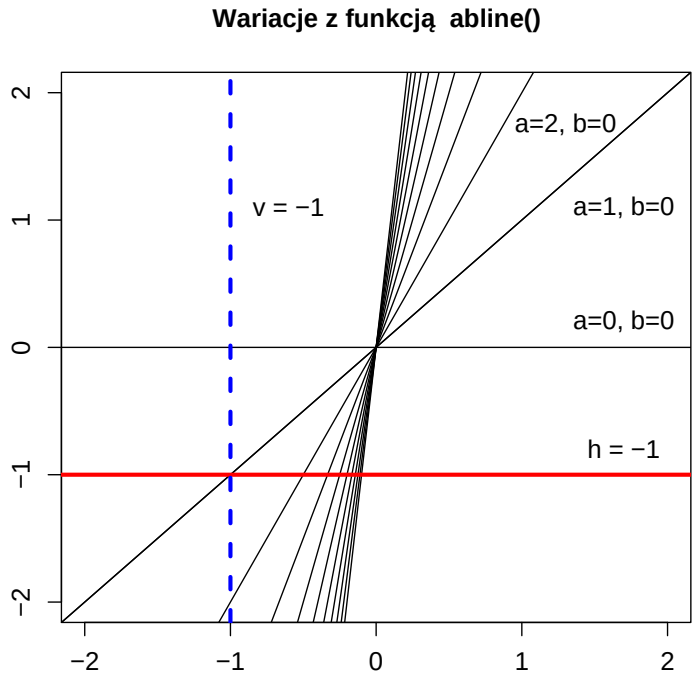
W trzeciej linii przykładu wartość `type="l"` oznacza, że rysowane mają być wyłącznie linie. Inne możliwe wartości tego argumentu, to **p** (punkty), **n** (nic), **s** (schodki), **h** (pionowe kreski, podobne do histogramu). Argument `col` funkcji `plot()` umożliwia wskazanie koloru w jakim ma być 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 4.3. Efekt działania powyższych poleceń znajduje się na rysunku 1.5.

Funkcje matematyczne można rysować korzystając również z funkcji `curve()`. Pierwszym argumentem jest wyrażenie (funkcja) zmiennej `x`, które ma być narysowane, kolejne dwa argumenty określają końce przedziału, na którym chcemy narysować to wyrażenie. Poniżej dwa przykłady do samodzielnego sprawdzenia.

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

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

Poniżej przedstawiono kilka przykładowych wywołań funkcji `abline()`. Wynik działania tego kodu jest umieszczony na rysunku 1.6. Za argument funkcji `abline()` można również podać model liniowy otrzymany z użyciem `lm(stats)`. W tym przypadku do wykresu dorysowana będzie prosta regresji (pozostawiam to czytelnikowi do samodzielnego sprawdzenia).



Rysunek 1.6: Przykładowe użycia funkcji *abline()*

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

# kilka prostych określonych przez równanie y = a x + b
abline(0,0)
abline(0,1)
for (i in 1:10)
         abline(0, i)
# dodajemy linię poziomą
abline(h=-1,lwd=3, col="red")
# dodajemy linię pionową
abline(v=-1,lwd=3, lty=2, col="blue")

# na wykres nanosimy kilka opisów
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")
```

Funkcja `text(graphics)` 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 4.3.

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, a więc 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 argumentach funkcji graficznych znaleźć można w podrozdziale 4.3.8.

1.6.5 Operacje 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(base)` i `setwd(base)`. 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 `File \ 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).

Aby wyświetlić listę plików znajdujących się w aktualnym (lub innym wskazanym) katalogu można posłużyć się funkcją `list.files(base)` lub `dir(base)`. 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.

```
> # funkcjami setwd() i getwd() można zmieniać katalog roboczy
> setwd("c:/Projekty/Dane")
> getwd()
[1] "c:/Projekty/Dane"
> # zobaczmy co jest w tym katalogu
> (list.files() -> lista.plikow)
[1] "daneFWF.txt"        "daneMatlab.MAT"     "daneMieszkania.csv"
[4] "daneSAS.sas7bdat"   "daneSoc.csv"        "daneSPSS.sav"
```

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

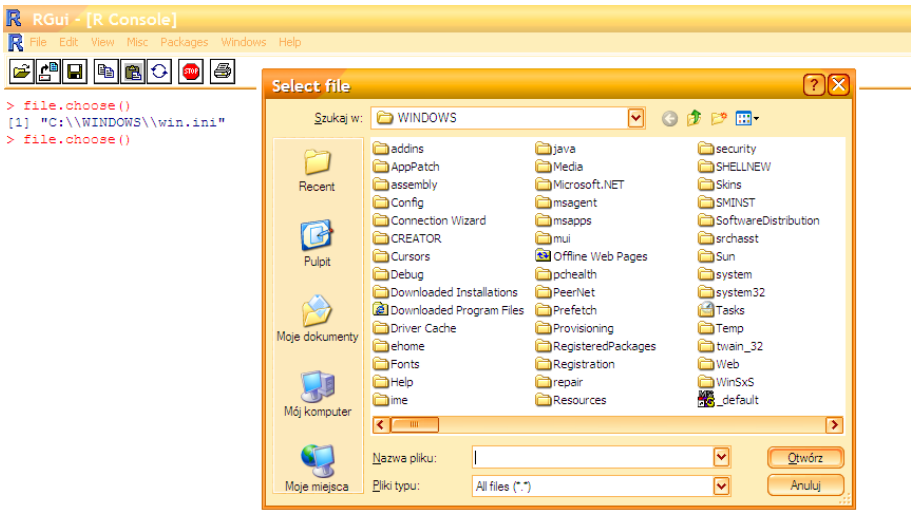
```
# generujemy losową nazwę dla tymczasowego pliku
tmpf <- tempfile()
# wpisujemy do tego pliku napis
cat(file=tmpf, "Poczatek pliku")
# tu operacje na pliku
# ....
# na koniec kasujemy tymczasowy plik
unlink(tmpf)
```

Kolejną przydatną funkcją do operacji na plikach jest `sink(base)`. Zapisuje ona do wskazanego pliku tekstowego przebieg interakcji z R. Jej wywołanie powoduje przekazanie wyjścia z konsoli 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 R.

Tabela 1.7: Funkcje do operacji na plikach

`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=F)`	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=T, recursive=F)`	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(base)` lub `choose.files(base)`. 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 R ma możliwość uzupełniania ścieżek. Jeżeli przy wpisywaniu ścieżki do pliku lub katalogu naciśniemy `Tab`, to R uzupełni nazwę wpisywanego katalogu lub pliku. Jeżeli ścieżkę można uzupełnić na wiele sposobów, to R wyświetla listę wszystkich możliwości.



Rysunek 1.7: Okno systemowe umożliwiające wybór jednego lub więcej plików. Wynik działania funkcji *file.choose()* lub *choose.files()*

1.6.5.1 Odczytywanie i zapisywanie plików tekstowych

W R jest kilka 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 odczytywanych 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 z pewnością zainteresowani oraz potrzebujący znajdą więcej informacji.

Analizując dane najczęściej korzysta się z danych tabelarycznych 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(base)` i `write.table(base)`. Obie funkcje są szczegółowo opisane w podrozdziale 2.3.1, 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 umieszczone informacje o nazwach kolumn (zmiennych) i/lub wierszy (przypadków). Kolejne wartości w pliku rozdzielane mogą być różnymi znakami (najpopularniejsze separatory to przecinek, średnik, spacja lub tabulacja). Do odczytanie danych z pliku, w którym w pierwszej linii umieszczone są nazwy kolumn a wartości kolejnych pól rozdzielane są znakami tabulacji można użyć polecenia:

```
macierz = read.table("nazwa.pliku", header=T, 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(base)`. Domyślnie wynik tej funkcji jest wyświetlany w konsoli, ale zmieniając wartość argumentu `file` możemy zapisać wyniki do pliku. Tak to wygląda na przykładzie:

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

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()`. Poniższy przykład zapisuje dane rozdzielając kolejne elementy znakiem tabulacji.

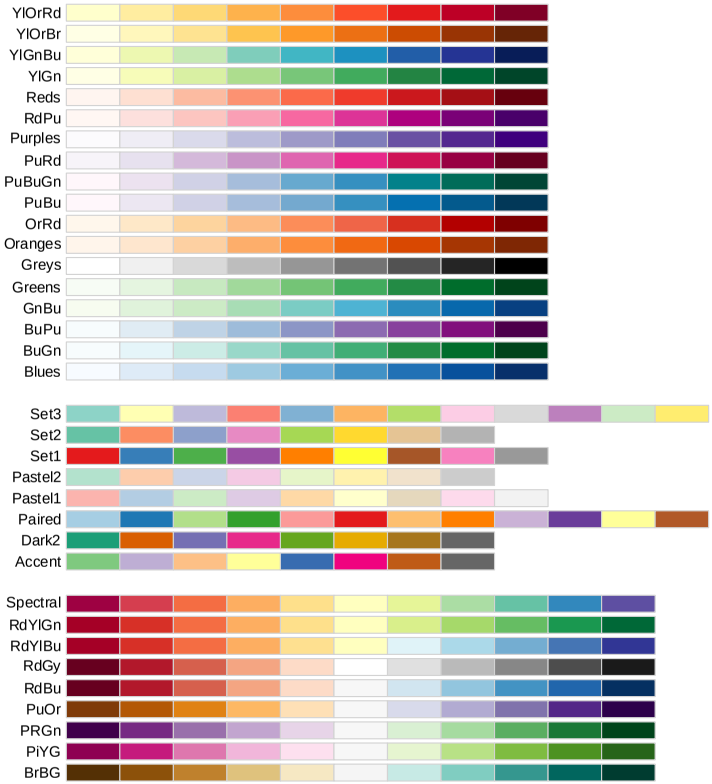
```
write.table(macierz, file="nazwa.pliku", sep="\t")
```

Duże i złożone obiekty lepiej przechowywać w postaci binarnej. Zapis w formacie plików binarnych umożliwia funkcja `save(base)`. Funkcja ta zapisuje wskazany obiekt lub listę obiektów w formacie `Rdata`. 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(base)`. Zapisuje ona do pliku wszystkie dostępne obiekty. Podobny efekt ma polecenie z menu `File / Save workspace`.

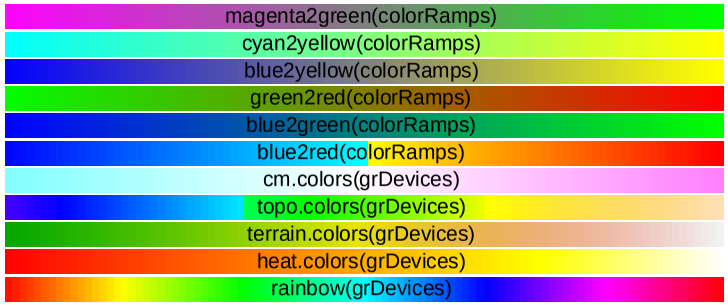
Binarna reprezentacja `RData` może różnić się dla różnych wersji R! Może więc się zdarzyć, że po zainstalowaniu nowej wersji R pewne pliki binarne nie odczytują się prawidłowo. Niestety w tym przypadku R nie sygnalizuje informacji o błędzie, użytkownik może być nawet nieświadomy źródła błędnych wyników. Z tego powodu, jeżeli chcemy przechowywać dane dłużej lub chcemy przenieść je na inny komputer (gdzie może być zainstalowana inna wersja R), to powinniśmy użyć innego formatu danych. Unikniemy dzięki temu przykrych niespodzianek w stylu „działało i już nie działa”.

Rysunek 4.37: Palety kolorów. Z przyczyn ekonomicznych prezentujemy wersję czarnobiałą. Kolor znaleźć można na stronie *http://www.biecek.pl/R/*

Kolory z pakietu RColorBrewer



Kolory z pakietów grDevices i colorRamps



Predefiniowane nazwy kolorów, colors()

coral3	deeppink4	gray27	gray87	grey39	grey99	lightpink1	mistyrose1	pink4	slategray1	
coral2	deeppink3	gray26	gray86	grey38	grey98	lightpink	mistyrose	pink3	slategray	
coral1	deeppink2	gray25	gray85	grey37	grey97	lightgrey	mintcream	pink2	slateblue4	
coral	deeppink1	gray24	gray84	grey36	grey96	lightgreen	midnightblue	pink1	slateblue3	yellowgreen
chocolate4	deeppink	gray23	gray83	grey35	grey95	lightgray	mediumvioletred	pink	slateblue2	yellow4
chocolate3	darkviolet	gray22	gray82	grey34	grey94	lightgoldenrodyello	mediumturquoise	peru	slateblue1	yellow3
chocolate2	darkturquoise	gray21	gray81	grey33	grey93	lightgoldenrod4	mediumspringgree	peachpuff4	slateblue	yellow2
chocolate1	darkslategrey	gray20	gray80	grey32	grey92	lightgoldenrod3	mediumslateblue	peachpuff3	skyblue4	yellow1
chocolate	darkslategray4	gray19	gray79	grey31	grey91	lightgoldenrod2	mediumseagreen	peachpuff2	skyblue3	yellow
chartreuse4	darkslategray3	gray18	gray78	grey30	grey90	lightgoldenrod1	mediumpurple4	peachpuff1	skyblue2	whitesmoke
chartreuse3	darkslategray2	gray17	gray77	grey29	grey89	lightgoldenrod	mediumpurple3	peachpuff	skyblue1	wheat4
chartreuse2	darkslategray1	gray16	gray76	grey28	grey88	lightcyan4	mediumpurple2	papayawhip	skyblue	wheat3
chartreuse1	darkslategray	gray15	gray75	grey27	grey87	lightcyan3	mediumpurple1	palevioletred4	sienna4	wheat2
chartreuse	darkslateblue	gray14	gray74	grey26	grey86	lightcyan2	mediumpurple	palevioletred3	sienna3	wheat1
cadetblue4	darkseagreen4	gray13	gray73	grey25	grey85	lightcyan1	mediumorchid4	palevioletred2	sienna2	wheat
cadetblue3	darkseagreen3	gray12	gray72	grey24	grey84	lightcyan	mediumorchid3	palevioletred1	sienna1	violetred4
cadetblue2	darkseagreen2	gray11	gray71	grey23	grey83	lightcoral	mediumorchid2	palevioletred	sienna	violetred3
cadetblue1	darkseagreen1	gray10	gray70	grey22	grey82	lightblue4	mediumorchid1	paleturquoise4	seashell4	violetred2
cadetblue	darkseagreen	gray9	gray69	grey21	grey81	lightblue3	mediumorchid	paleturquoise3	seashell3	violetred1
burlywood4	darksalmon	gray8	gray68	grey20	grey80	lightblue2	mediumblue	paleturquoise2	seashell2	violetred
burlywood3	darkred	gray7	gray67	grey19	grey79	lightblue1	mediumaquamarin	paleturquoise1	seashell1	violet
burlywood2	darkorchid4	gray6	gray66	grey18	grey78	lightblue	maroon4	paleturquoise	seashell	turquoise4
burlywood1	darkorchid3	gray5	gray65	grey17	grey77	lemonchiffon4	maroon3	palegreen4	seagreen4	turquoise3
burlywood	darkorchid2	gray4	gray64	grey16	grey76	lemonchiffon3	maroon2	palegreen3	seagreen3	turquoise2
brown4	darkorchid1	gray3	gray63	grey15	grey75	lemonchiffon2	maroon1	palegreen2	seagreen2	turquoise1
brown3	darkorchid	gray2	gray62	grey14	grey74	lemonchiffon1	maroon	palegreen1	seagreen1	turquoise
brown2	darkorange4	gray1	gray61	grey13	grey73	lemonchiffon	magenta4	palegreen	seagreen	tomato4
brown1	darkorange3	gray0	gray60	grey12	grey72	lawngreen	magenta3	palegoldenrod	sandybrown	tomato3
brown	darkorange2	gray	gray59	grey11	grey71	lavenderblush4	magenta2	orchid4	salmon4	tomato2
blueviolet	darkorange1	goldenrod4	gray58	grey10	grey70	lavenderblush3	magenta1	orchid3	salmon3	tomato1
blue4	darkorange	goldenrod3	gray57	grey9	grey69	lavenderblush2	magenta	orchid2	salmon2	tomato
blue3	darkolivegreen4	goldenrod2	gray56	grey8	grey68	lavenderblush1	linen	orchid1	salmon1	thistle4
blue2	darkolivegreen3	goldenrod1	gray55	grey7	grey67	lavenderblush	limegreen	orchid	salmon	thistle3
blue1	darkolivegreen2	goldenrod	gray54	grey6	grey66	lavender	lightyellow4	orangered4	saddlebrown	thistle2
blue	darkolivegreen1	gold4	gray53	grey5	grey65	khaki4	lightyellow3	orangered3	royalblue4	thistle1
blanchedalmond	darkolivegreen	gold3	gray52	grey4	grey64	khaki3	lightyellow2	orangered2	royalblue3	thistle
black	darkmagenta	gold2	gray51	grey3	grey63	khaki2	lightyellow1	orangered1	royalblue2	tan4
bisque4	darkkhaki	gold1	gray50	grey2	grey62	khaki1	lightyellow	orangered	royalblue1	tan3
bisque3	darkgrey	gold	gray49	grey1	grey61	khaki	lightsteelblue4	orange4	royalblue	tan2
bisque2	darkgreen	ghostwhite	gray48	grey0	grey60	ivory4	lightsteelblue3	orange3	rosybrown4	tan1
bisque1	darkgray	gainsboro	gray47	grey	grey59	ivory3	lightsteelblue2	orange2	rosybrown3	tan
bisque	darkgoldenrod4	forestgreen	gray46	greenyellow	grey58	ivory2	lightsteelblue1	orange1	rosybrown2	steelblue4
beige	darkgoldenrod3	floralwhite	gray45	green4	grey57	ivory1	lightsteelblue	orange	rosybrown1	steelblue3
azure4	darkgoldenrod2	firebrick4	gray44	green3	grey56	ivory	lightslategrey	olivedrab4	rosybrown	steelblue2
azure3	darkgoldenrod1	firebrick3	gray43	green2	grey55	indianred4	lightslategray	olivedrab3	red4	steelblue1
azure2	darkgoldenrod	firebrick2	gray42	green1	grey54	indianred3	lightsteblue	olivedrab2	red3	steelblue
azure1	darkcyan	firebrick1	gray41	green	grey53	indianred2	lightskyblue4	olivedrab1	red2	springgreen4
azure	darkblue	firebrick	gray40	gray100	grey52	indianred1	lightskyblue3	olivedrab	red1	springgreen3
aquamarine4	cyan4	dodgerblue4	gray39	gray99	grey51	indianred	lightskyblue2	oldlace	red	springgreen2
aquamarine3	cyan3	dodgerblue3	gray38	gray98	grey50	hotpink4	lightskyblue1	navyblue	purple4	springgreen1
aquamarine2	cyan2	dodgerblue2	gray37	gray97	grey49	hotpink3	lightskyblue	navy	purple3	springgreen
aquamarine1	cyan1	dodgerblue1	gray36	gray96	grey48	hotpink2	lightseagreen	navajowhite4	purple2	snow4
aquamarine	cyan	dodgerblue	gray35	gray95	grey47	hotpink1	lightsalmon4	navajowhite3	purple1	snow3
antiquewhite4	cornsilk4	dimgrey	gray34	gray94	grey46	hotpink	lightsalmon3	navajowhite2	purple	snow2
antiquewhite3	cornsilk3	dimgray	gray33	gray93	grey45	honeydew4	lightsalmon2	navajowhite1	powderblue	snow1
antiquewhite2	cornsilk2	deepskyblue4	gray32	gray92	grey44	honeydew3	lightsalmon1	navajowhite	plum4	snow
antiquewhite1	cornsilk1	deepskyblue3	gray31	gray91	grey43	honeydew2	lightsalmon	moccasin	plum3	slategrey
antiquewhite	cornsilk	deepskyblue2	gray30	gray90	grey42	honeydew1	lightpink4	mistyrose4	plum2	slategray4
aliceblue	cornflowerblue	deepskyblue1	gray29	gray89	grey41	honeydew	lightpink3	mistyrose3	plum1	slategray3
white	coral4	deepskyblue	gray28	gray88	grey40	grey100	lightpink2	mistyrose2	plum	slategray2

Zbiory danych

W tym miejscu przedstawimy zbiory danych wykorzystywane w poprzednich rozdziałach. Są one umieszczone w Internecie i można je ściągnąć podanymi poniżej poleceniami. Jeżeli nie mamy własnych danych, to warto na tych przećwiczyć omawiane w tej książce funkcje.

4.4 Zbiór danych onkologicznych: daneO.csv

Ten zbiór danych zawiera informacje o dziewięciu zmiennych zmierzonych dla 97 pacjentek z wrocławskiego oddziału onkologii. Dane zostały przetworzone tak, by nie można było wydobyć z nich jakichkolwiek, nawet częściowych informacji o pacjentkach. Dane te nadają się do ćwiczeń, ale z uwagi na zbyt małą liczbę przypadków nie mogą być wykorzystane do wnioskowania o prawdziwych skutkach tej choroby.

Patrick Burns: In the old days with S-PLUS, the rule of thumb was that you needed 10 times as much memory as your dataset. [...] R (and current versions of S-PLUS) are more frugal than S-PLUS was back then.
Ajay Shah: Hmm, so it'd be interesting to apply current prices of RAM and current prices of R, to work out the break-even point at which it's better tobuy SAS! :-) Without making any calculations, I can't see how SAS can compete with the price of 4G of RAM.

Patrick Burns, Ajay Shah
fortune(93)

```
> daneO = read.table("http://www.biecek.pl/R/dane/daneO.csv", sep=";", h=T)
> summary(daneO)
      Wiek         Rozmiar.guza    Wezly.chlonne      Nowotwor
 Min.   :29.00   Min.   :1.000   Min.   :0.0000   Min.   : 1.000
 1st Qu.:42.00   1st Qu.:1.000   1st Qu.:0.0000   1st Qu.: 2.000
 Median :46.00   Median :1.000   Median :0.0000   Median : 2.000
 Mean   :45.52   Mean   :1.268   Mean   :0.4124   Mean   : 2.221
 3rd Qu.:50.00   3rd Qu.:2.000   3rd Qu.:1.0000   3rd Qu.: 3.000
 Max.   :57.00   Max.   :2.000   Max.   :1.0000   Max.   : 3.000
                                                  NA's   :11.000
 Receptory.estrogenowe Receptory.progesteronowe Niepowodzenia Okres.bez.wznowy
 (-)  :37              (-)  :24                 brak  :84      Min.   :10.00
 (+)  :21              (+)  :18                 wznowa:13      1st Qu.:30.75
 (++) :24              (++) :32                                Median :38.00
 (+++): 9              (+++):16                                Mean   :37.41
 NA's : 6              NA's : 7                                3rd Qu.:45.00
                                                               Max.   :54.00
                                                               NA's   : 1.00

      VEGF
 Min.   :  118
 1st Qu.:  629
 Median : 1489
 Mean   : 2627
 3rd Qu.: 3240
 Max.   :22554
```

4.5 Zbiór danych o cenach mieszkań: daneMieszkania.csv

Ten zbiór danych zawiera informacje o pięciu parametrach dla 200 mieszkań z wrocławskiego rynku nieruchomości (ceny z roku 2004 a więc już trochę nieaktualne, w każdym razie nie mogą być traktowane jako oferta handlowa).

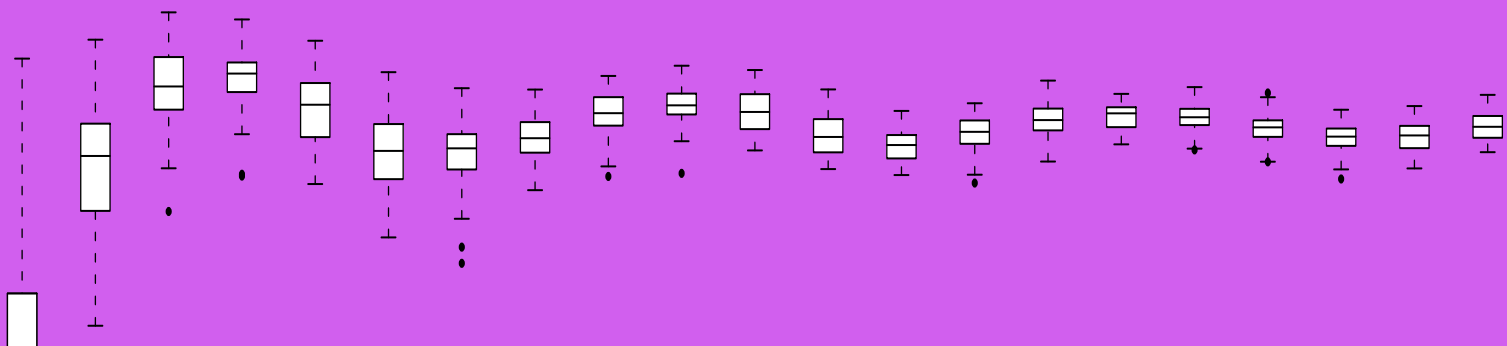
```
> mieszkania = read.table("http://www.biecek.pl/R/dane/daneMieszkania.csv",
    header=T,sep=";")
> summary(mieszkania)
      cena            pokoi        powierzchnia       dzielnica     typ.budynku
 Min.   : 83280   Min.   :1.00   Min.   :17.00   Biskupin   :65   kamienica :61
 1st Qu.:143304   1st Qu.:2.00   1st Qu.:31.15   Krzyki     :79   niski blok:63
 Median :174935   Median :3.00   Median :43.70   Srodmiescie:56   wiezowiec :76
 Mean   :175934   Mean   :2.55   Mean   :46.20
 3rd Qu.:208741   3rd Qu.:3.00   3rd Qu.:61.40
 Max.   :295762   Max.   :4.00   Max.   :87.70
```

4.6 Zbiór danych socjodemograficznych: daneSoc.csv

Ten zbiór danych zawiera informacje o siedmiu zmiennych (głównie socjodemograficznych) zmierzonych dla 204 pacjentów jednej z wrocławskich przychodni.

```
> daneSoc = read.csv("http://www.biecek.pl/R/dane/daneSoc.csv",sep=";")
> summary(daneSoc)
      wiek          wyksztalcenie     st.cywilny         plec
 Min.   :22.00   podstawowe:93    singiel  :120   kobieta  : 55
 1st Qu.:30.00   srednie   :55    w zwiazku: 84   mezczyzna:149
 Median :45.00   wyzsze    :34
 Mean   :43.16   zawodowe  :22
 3rd Qu.:53.00
 Max.   :75.00
                 praca     cisnienie.skurczowe cisnienie.rozkurczowe
 nie pracuje      : 52   Min.   : 93.0       Min.   : 57.00
 uczen lub pracuje:152   1st Qu.:126.0       1st Qu.: 77.00
                         Median :137.5       Median : 80.00
                         Mean   :137.0       Mean   : 80.43
                         3rd Qu.:148.0       3rd Qu.: 86.25
                         Max.   :178.0       Max.   :107.00
```

Skorowidz



Przewodnik po pakiecie $\mathcal{R}$

Pakiet R jest wykorzystywanym na całym świecie narzędziem do analizy danych, zarówno finansowych, biologicznych, medycznych jak i dowolnych innych. Możliwościami R przewyższa większość profesjonalnych pakietów statystycznych. Rozwijany przez największych światowych ekspertów oraz setki entuzjastów umożliwia przeprowadzenie rzetelnej analizy, zobrazowanie wyników czytelnymi wykresami, automatyczne wygenerowanie raportu, wysłanie go mailem renderując przy okazji trójwymiarową animację. Na dodatek R jest darmowy do wszelkich zastosowań, tak edukacyjnych jak i biznesowych.

„Przewodnik...” to pierwsza polskojęzyczna książka w całości poświęcona R. Czytelnik znajdzie tu przystępne wprowadzenie, opis użytecznych bibliotek, zaawansowanych mechanizmów oraz informacje na temat setek przydatnych funkcji. Szczególny nacisk został położony na przedstawienie elastyczności języka R oraz zademonstrowanie statystycznych i graficznych możliwości tego pakietu.

Z „Przewodnika...” skorzystać może szeroka rzesza użytkowników R:

- Osoby początkujące, nie mające jeszcze kontaktu z R, znajdą tu łagodne wprowadzenie, wiele komentarzy, opisów i szczegółowo omówionych przykładów.
- Osoby korzystające już z R znajdą opis wielu przydatnych dodatkowych zagadnień takich jak programowanie objaśniające, debugger, profiler, leniwa ewaluacja, odczytywanie danych z rożnych źródeł danych i wielu innych. Pozwoli im to usystematyzować, uzupełnić i pogłębić wiedzę o pakiecie R.
- Eksperci, pracujący z R na co dzień, znajdą tu podręczną ściągawkę zawierającą 43 tabele, 98 ilustracji, 332 przykłady kodu R oraz opis 578 przydatnych funkcji, często wraz z listą i opisem argumentów oraz dodatkowymi uwagami. Zamieszczone schematy i ilustracje pozwolą na szybkie wyszukiwanie ważnych informacji.

Wydanie książki dofinansowane przez
Instytut Podstaw Informatyki PAN