Marek Gągolewski

WYDANIE II
POSZERZONE

Programowanie w języku R

Analiza danych, obliczenia, symulacje

PWN

Programowanie w języku R

Marek Gągolewski

Programowanie w języku R

Analiza danych, obliczenia, symulacje

PWN

Projekt okładki **Hubert Zacharski**

Ilustracja na okładce **shutterstock/antishock**

Wydawca **Łukasz Łopuszański**

Redaktor prowadzący **Iwona Lewandowska**

Redaktor **Ewa Ławrynowicz**

Koordynator produkcji **Anna Bączkowska**

Skład i łamanie **FixPoint**, Warszawa

ISBN 978-83-01-18939-6

Wydanie II
Warszawa 2016

Wydawnictwo Naukowe PWN SA
02-460 Warszawa, ul. Gottlieba Daimlera 2
tel. 22 69 54 321, faks 22 69 54 288
infolinia 801 33 33 88
e-mail: pwn@pwn.com.pl; reklama@pwn.pl
www.pwn.pl

Druk i oprawa: OSDW Azymut Sp. z o.o.

SPIS TREŚCI

PRZEDMOWA

Korporacje, rządy, instytuty naukowe, a nawet zwykli użytkownicy internetu generują informacje różnego rodzaju: dotyczące rozrywki, kultury, komunikacji międzyludzkiej, ekonomii, handlu, przemysłu, transportu, zdrowia, środowiska itd. Postęp technologiczny, jaki dokonał się w naukach informacyjnych na przestrzeni ostatnich dekad, skutkuje tym, że dzisiejszy świat wytwarza więcej danych, niż jest ich w stanie efektywnie przetworzyć. Owo „wąskie gardło" często wcale nie jest spowodowane brakiem dostępności odpowiednich algorytmów i narzędzi, ale raczej trudnością w znalezieniu – tak przecież rozchwytywanych przez pracodawców – dobrze przygotowanych profesjonalistów.

Przekształcanie surowych danych (ilościowych, jakościowych, tekstu itp.) na przyswajalną *wiedzę* jest celem działania m.in. *analityków danych*, specjalistów *business intelligence*, *badaczy opinii i rynku* czy wreszcie *data scientists*. Dziedziny te wymagają nie tylko obycia w zakresie szeroko pojętej probabilistyki, szeregów czasowych, metod statystycznego i maszynowego uczenia się, modelowania, przetwarzania obrazów, językoznawstwa itp., co wystarczająco pilny student może poznać lepiej lub gorzej z różnego rodzaju kursów lub lektury dostępnych podręczników, lecz także wyobraźni, kreatywności, komunikatywności i znajomości szczególnych obszarów zastosowań (np. procesów zachodzących w firmie, ekonomii, medycyny, nauk społecznych). I wreszcie, za wręcz niezbędną uważa się *biegłą umiejętność obsługi narzędzi informatycznych, które służą do przechowywania i przetwarzania informacji, wydobywania z nich wiedzy i prezentacji uzyskanych wyników* – i to na poziomie, który nie tyle implikuje sprawne wykorzystanie istniejących rozwiązań, ile przede wszystkim umożliwia *analizę, projektowanie, implementację, testowanie i wdrażanie własnych pomysłów*, a także dzielenie się efektami pracy z innymi.

R a inne programy. Na rynku znajdziemy szeroki wybór oprogramowania, które wspiera pracę analityków danych. Przede wszystkim mamy dostęp do narzędzi skrojonych na miarę potrzeb tej grupy osób, przez co praca w nich jest bardzo wygodna i podstawowe czynności można wykonać przy ich użyciu naprawdę bardzo efektywnie. Należą do nich m.in. SAS, Stata, Statistica, SPSS i Weka.

Jednakże w praktyce to, co jest ich zaletą, może szybko obrócić się w przypadku konieczności zmierzenia się z bardziej ambitnymi, nieszablonowymi zadaniami w wadę

– a takich wyzwań jest coraz więcej. Pomimo dość stromej, w każdym razie początkowo, „krzywej uczenia się" środowiska programistyczne, których trzonem są interpretowane języki programowania ogólnego zastosowania, oferują tu znacznie większy stopień swobody, przez co możliwości ich użycia są praktycznie nieograniczone. Wśród najbardziej popularnych narzędzi tego typu znajdziemy m.in. chętnie wybierany przez „rasowych" programistów język Python z rodziną pakietów SciPy, SciKits i Pandas, zob. [29], oraz szczególnie cenione przez statystyków, osoby zajmujące się wizualizacją danych i analityków danych z mniejszym informatycznym przygotowaniem, środowisko R [73].

Środowisko R, będące głównym bohaterem niniejszej książki, jest aktywnie rozwijane przez społeczność *open source*, której każdy z nas może stać się czynnym członkiem. Dzięki wytężonej pracy licznych pasjonatów i systemowi ogólnie dostępnych pakietów[1] (ang. *packages*) niezmiernie łatwe jest rozbudowywanie zdolności tego narzędzia o tak istotne możliwości jak komunikacja z bazami danych, przetwarzanie wielkich zbiorów danych, obliczenia równoległe, zbieranie danych ze stron internetowych itd. Mamy tutaj do wyboru mnóstwo implementacji znanych technik i metod statystycznej analizy danych oraz algorytmów maszynowego uczenia się, m.in. analizy przeżycia, algorytmów *bootstrapowych*, sieci neuronowych, metod analizy regresji i klasyfikacji, analizy skupień, szeregów czasowych, danych finansowych, redukcji wymiarowości itd. Co więcej, przeglądając publikacje naukowe, możemy zaobserwować, że wiele nowych metod analizy danych jest najpierw implementowanych właśnie w R.

CIEKAWOSTKA

Dużym niedopatrzeniem jest określanie R mianem „pakietu do obliczeń statystycznych". Taka etykieta działa wręcz na jego szkodę, kreuje bowiem fałszywą opinię, że jest to narzędzie o wąskich możliwościach. Po pierwsze, jest to całe *środowisko* zbudowane wokół funkcyjnego, interpretowanego języka programowania ogólnego zastosowania o naprawdę ciekawych i godnych podziwu cechach – obliczenia statystyczne i wizualizacja danych są tylko jednym z wielu możliwych obszarów jego użycia, aczkolwiek oczywiście najbardziej docenianym na całym świecie. Po drugie, to właśnie dzięki *pakietom* (nakładkom rozszerzającym jego możliwości) obliczenia takowe są możliwe. Pakietów służących do przeprowadzania „obliczeń statystycznych" (np. stats) nie trzeba wcale ładować – zamiast nich można korzystać z narzędzi zupełnie innego rodzaju (por. też tematykę najczęściej pobieranych pakietów na s. 420).

Cel książki. Mówi się, że komputer jest tak mądry, jak jego programista. Dobry programista to taki, który potrafi zaprogramować wszystko, o czym jest w stanie pomyśleć. Nabywszy solidnych umiejętności, dostrzega, że często nie warto jest „wyważać otwartych drzwi" i można skorzystać z wyników pracy innych osób. Jest przygotowany jednak do tego, by dostrzec alternatywne podejścia do interesującego go zagadnienia, potrafi docenić ich mocne strony i dostrzec ograniczenia. Wreszcie wie, kiedy należy „wziąć

[1] W repozytorium CRAN znajdziemy aktualnie (czerwiec 2016 r.) prawie 10 000 pakietów. Co więcej, obserwujemy, że liczba pakietów rośnie wykładniczo.

sprawy w swoje ręce" i pewne mechanizmy dostosować samodzielnie do własnych, konkretnych potrzeb.

Znakomita część publikacji dostępnych nie tylko na polskim (por. np. [8, 9, 18, 34, 36, 77, 90, 99]), ale i zagranicznym rynku wydawniczym dotyczy sposobów wykorzystywania środowiska R w różnych zastosowaniach praktycznych – czyli uczy tak naprawdę *obcowania* z konkretnymi *pakietami*. Monografii na temat samego programowania w języku R jest niewiele. Oprócz napisanych przez jego twórców kilku pozycji, w szczególności [5, 14, 15, 89], w których są przede wszystkim omówione wybrane aspekty tworzenia oprogramowania i które są raczej przeznaczone dla tych, którzy znają już ten język dość dobrze, mamy jeszcze np. dość przystępnie napisane angielskojęzyczne książki [50, 56, 93]. Żadna z nich nie omawia jednak zarówno *fundamentów* języka, jak i zagadnień zaawansowanych w sposób wystarczająco wyczerpujący.

WAŻNE

Niniejsza publikacja stanowi kompletny i wyczerpujący kurs programowania w języku R – od omówienia *podstaw języka*, aż po przedstawienie *zaawansowanych zagadnień*. Towarzyszące jej ćwiczenia pomogą rozwijać umiejętności biegłego implementowania nowych algorytmów, dostosowywania do własnych potrzeb już istniejących oraz automatyzowania żmudnych – gdyby je wykonywać ręcznie – zadań w dosłownie każdym obszarze zastosowań. Co więcej, lektura jej pozwoli zrozumieć, dlaczego obliczenia w R są realizowane w taki, a nie inny sposób.

Szczególna uwaga została poświęcona sposobom ładowania i zapisywania zbiorów danych oraz ich generowaniu na podstawie różnych źródeł, a także ich wstępnej obróbce, czyszczeniu, aż po samą analizę i prezentację wyników. Ma ona za zadanie wspomagać nas w drodze ku programistycznej samodzielności – abyśmy mogli wyjść poza gotowe schematy i śmiało mierzyć się z nowymi wyzwaniami, przed którymi stawia nas tzw. era informacji. Odpowiedni nacisk został też położony na tworzenie kodu wysokiej jakości, nie tylko w sensie jego poprawności, czytelności i łatwości rozbudowy, ale i szybkości działania oraz ekonomicznego zużywania pamięci.

Omawiając zagadnienia programowania w języku R na słusznym poziomie ogólności, zrezygnowaliśmy ze „słownikowego" przeglądu wielu ciekawych funkcji rozproszonych po różnych pakietach (algorytmów weryfikacji hipotez, analizy wariancji i przeżycia, drzew klasyfikacyjnych, maszyn wektorów podpierających, szczególnych typów wykresów – lista ta mogłaby się ciągnąć w nieskończoność). Obiecujemy jednak, że po lekturze tej książki będziemy w stanie znaleźć potrzebne nam narzędzie i samodzielnie je poznać, a w razie potrzeby napisać własne. Dzięki temu zmniejszyliśmy też ryzyko tego, że dzieło to będzie po prostu nie tylko nudne, ale i szybko się zdezaktualizuje.

Adresaci książki. Idealnego Czytelnika tej książki najłatwiej możemy określić przez… zaprzeczenie. Otóż nie polecamy jej osobom, które:

— wcale nie są zainteresowane tym, by nauczyć się programować w języku R, ani tym, by udoskonalić swoje umiejętności;

— nie mają potrzeby w swojej pracy zawodowej lub naukowej wykonywać żadnych obliczeń, automatyzować procesów przetwarzania danych ani tworzyć grafiki użytkowej (np. wykresów);
— nie mają czasu przeczytać jej od początku do końca;
— znają już język R tak (nie)dobrze, że wydaje im się, iż niczego nowego się nie dowiedzą z lektury rozdz. 2–9;
— nie zamierzają twórczo poeksperymentować z przedstawionymi przykładami oraz nie mają chęci rozwiązywania ćwiczeń;
— boją się „zarazić" radością z tworzenia nowego oprogramowania.

W przeciwnym przypadku niniejsza pozycja jest po prostu idealna. Innymi słowy:

WAŻNE

Pozycję tę możemy polecić każdemu, kto jest zainteresowany dokładnym poznaniem wszystkich możliwości języka i środowiska R, a także efektywnym wykorzystaniem go w swojej pracy zawodowej lub naukowej. Dotyczy to zarówno osób, które nie programowały jeszcze w R, jak i tych, które z poziomu podstawowego, średniozaawansowanego lub zaawansowanego chcą wspiąć się jeszcze wyżej.

Układ książki. Przedstawiony w książce materiał jest podzielony na pięć następujących bloków tematycznych:

1) W pierwszej części przedstawiamy elementarz programowania w R oraz najważniejsze operacje na podstawowych typach danych, w tym na wektorach atomowych, funkcjach i listach. Tłumaczymy, dlaczego w R należy unikać stosowania m.in. pętli i jak się bez nich obyć. Mówimy też o atrybutach obiektów i ich roli w programowaniu zorientowanym obiektowo w stylu S3 oraz o tym, że typy złożone, takie jak macierz, czynnik czy ramka danych, są prostym rozszerzeniem typów podstawowych. Przedstawimy najważniejsze techniki pielęgnowania kodu i zapewniania, że będzie on dobrej jakości, po czym wprowadzimy wyrażenia służące do samodzielnej kontroli przepływu sterowania, które przydają się w sytuacjach, gdy jakiegoś algorytmu naprawdę nie da się zaimplementować wyłącznie przy użyciu funkcji wbudowanych.
2) W drugiej części omawiamy sposoby przetwarzania danych tekstowych, odczytywania i zapisywania najważniejszych formatów plików, zbierania danych ze stron internetowych oraz dostępu do baz danych SQL.
3) Trzecią część poświęcamy prezentacji wyników obliczeń, w tym tworzeniu dowolnie skomplikowanych wykresów i raportów.
4) W czwartej części doskonalimy nasze umiejętności w konkretnych obszarach zastosowań: analizie danych, obliczeniach numerycznych i naukowych oraz symulacjach.
5) W piątej części poznajemy zaawansowane aspekty programowania w języku R, takie jak środowiskowy model obliczeń, parsowanie i niestandardową ewaluację wyrażeń oraz mechanizmy programowania zorientowanego obiektowo przy

użyciu klas typu S3, S4 i RC. Mówimy także, jak tworzyć rozszerzenia dla środowiska R w języku C++, która to umiejętność przydaje się szczególnie w sytuacjach, gdy pewne funkcje okazują się być „wąskim gardłem" pod względem wydajności lub zużycia pamięci.

Zmiany w drugim wydaniu. Pierwsze wydanie *Programowania w języku R* zostało przygotowane w 2013 r. W ukazującym się po 3 latach drugim wydaniu zaszedł szereg istotnych zmian – wszystkie partie tekstu zostały zrewidowane, a wiele z nich nawet zostało napisanych ponownie. Do najbardziej zauważalnych różnic możemy zaliczyć:

- uproszczenie materiału przedstawianego na początku każdego rozdziału – trudniejsze partie zostały przesunięte zgodnie z duchem „od ogółu do szczegółu";
- dodanie ćwiczeń (wraz z rozwiązaniami) do tekstu głównego;
- zamieszczenie większej liczby przykładów z analizy danych (opis implementacji wielu ważnych metod maszynowego uczenia się, więcej uwagi poświęconej ramkom danych);
- nowy rozdział na temat integracji C++ z R (rozdz. 20; coraz więcej pakietów jest tworzona przy użyciu Rcpp, co bardzo pozytywnie wpłynęło w ostatnich latach na szeroko pojętą jakość tego rodzaju dodatków);
- omówienie pakietu stringi w rozdziale na temat przetwarzania napisów (rozdz. 10);
- przesunięcie rozdziału na temat pętli najdalej, jak to było możliwe;
- opis sposobów dostępu do baz danych SQL, przetwarzania plików JSON i XML oraz technik wydobywania informacji ze stron internetowych (*web scraping*) w rodz. 11;
- złączenie dwóch rozdziałów na temat generowania grafiki w jeden (rozdz. 12);
- opis języka Markdown w rozdz. 13 (jego popularność i dostępność z poziomu R w ostatnim czasie znacząco wzrosła);
- dokładniejsze omówienie sposobów tworzenia pakietów i poprawy jakości kodu w rozdz. 8;
- bardziej przystępne omówienie mechanizmów niestandardowej ewaluacji i jej znaczenia w rozdz. 17 i 18.

Uwagi o użytym oprogramowaniu. Wszystkie obliczenia zostały wykonane przy użyciu środowiska R w wersji deweloperskiej 3.4.0 (SVN rev. 70486) oraz pakietu knitr (por. [97] i rozdz. 13), a sama książka została złożona w systemie LaTeX [47, 68].

Podziękowania. Dziękuję moim przyjaciołom i bliskim współpracownikom za liczne uwagi i komentarze na różnych etapach przygotowywania pierwszego i/lub drugiego wydania, w szczególności Maciejowi Bartoszukowi, Annie Cenie, Janowi Laskowi, dr Annie Olwert, Agacie Sakowicz, Bartłomiejowi Tartanusowi, Barbarze Żogale-Siudem oraz p. red. Izabeli Mika. Składam także wyrazy podziękowania dla moich studentów z Wydziału Matematyki i Nauk Informacyjnych Politechniki Warszawskiej, uczestnikom i organizatorom kursów *Data Science Retreat* w Berlinie i *International PhD Studies Program* w Instytucie Podstaw Informatyki PAN, a także wszystkim użytkownikom

pakietów, których jestem autorem lub współautorem. Inspirujecie mnie do ciągłego doskonalenia moich umiejętności.

Zdaję sobie sprawę, że dzieło takiej objętości nie jest niestety pozbawione błędów, za które całkowitą odpowiedzialność ponoszę oczywiście ja sam. Proszę więc o zgłaszanie wszelkich uwag na mój adres e-mail: marek@gagolewski.com. Na stronie internetowej github.com/gagolews/Programowanie_w_jezyku_R/, do której odwiedzenia zachęcam, zamieściłem przykładowe kody źródłowe i aktualną wersję erraty.

Marek Gągolewski

Warszawa, czerwiec 2016 r.

Część I

PODSTAWY

ŚRODOWISKO R I PROGRAM RSTUDIO 1

R jest wolnym (otwartym i darmowym), zaawansowanym środowiskiem oraz językiem programowania cenionym na świecie przede wszystkim za możliwości przeprowadzania w nim *odtwarzalnych* (ang. *reproducible*) obliczeń statystycznych i numerycznych, analizy danych oraz tworzenia raportów i wysokiej jakości grafiki.

Środowisko R powstało w 1997 r. na Uniwersytecie w Auckland w Nowej Zelandii. Jest ono otwartą alternatywą dla używanego jeszcze przez niektóre instytucje komercyjnego pakietu S-PLUS, zaprojektowanego w laboratoriach Bella przez Johna Chambersa i jego kolegów. Rozwija się bardzo dynamicznie: kolejne wydania, uwzględniające nowe funkcje i poprawki starych błędów, pojawiają się kilka razy w roku. Pierwsza „nietestowa" wersja 1.0 została wydana w 2000 r., a jego dojrzałość potwierdza wydana w 2013 r. wersja 3.0. Jądro R składa się z napisanej w językach C i Fortran implementacji znanego od 1976 r. i wciąż rozwijanego języka S[1].

WAŻNE

Licencja GNU GPL (ang. *General Public License*) zezwala na wykorzystywanie środowiska R także w zastosowaniach komercyjnych.

Na marginesie, środowisko to jest czasem nazywane GNU S, by podkreślić jego *otwartość*. Istnieje także jego komercyjna wersja, zoptymalizowana pod kątem wysoko wydajnych obliczeń na dużych zbiorach danych – Microsoft R.

1.1. Cechy języka i środowiska R

Języków programowania jest wiele i, co warte podkreślenia, żaden z nich nie jest idealny. Niektóre języki są jednak uznawane za szczególnie godne uwagi w określonych zastosowaniach. Zawsze warto znać nie tylko mocne strony, ale i ograniczenia używanego przez siebie narzędzia, aby móc lepiej wykorzystać je w praktyce.

Składnia R przypomina tę stosowaną w C i C++, jednak jego semantyka jest zbliżona do funkcyjnego Scheme, por. [1].

[1] Dlatego programistów języka R może także zainteresować literatura dedykowana S.

Następujące cechy języka R są warte odnotowania.

— Jest językiem *ogólnego zastosowania* – można w nim zaimplementować praktycznie każdy algorytm. Odróżnia go to od języków szczególnego zastosowania, takich jak np. SQL.
— Jest przeznaczony raczej do pisania *„małych" programów*, w których ważne jest sedno, a nie „otoczka". Skupiamy się tutaj najczęściej na samych obliczeniach, a nie np. na sposobach interakcji z użytkownikiem, którym najczęściej jest bowiem ściśle określony specjalista.
— R cechuje się bardzo *zwięzłą składnią*, tzn. mało kodu daje duży, dość złożony rezultat. Owa ekspresywność znacząco ułatwia programowanie.
— Jest *językiem interpretowanym*. Dzięki temu można pracować w nim w sposób interaktywny, prawie natychmiast otrzymując wynik wykonywanych poleceń. Doskonale ułatwia to wytwarzanie prototypów implementacji ciekawych algorytmów (ang. *rapid prototyping*).

  Mimo że przez to programy tworzone w ten sposób będą nieco mniej wydajne od języków kompilowanych, w R możemy łatwo odwoływać się do gotowego, skompilowanego kodu pochodzącego z zewnętrznych, dynamicznie ładowanych bibliotek. Z tego powodu często twórcy pakietów R decydują się na tworzenie newralgicznych fragmentów kodu w języku C lub C++, np. przy użyciu pakietu Rcpp (rozdz. 20), by zapewnić użytkownikom końcowym dobrze zoptymalizowane pod względem czasochłonności rozwiązania.

Należy także zwrócić uwagę, że interesujący nas język jest składnikiem większego „ekosystemu" – z tego też powodu bardzo często mówimy o całym *środowisku R*.

— Ma ono dość rozbudowane *możliwości generowania wysokiej jakości grafiki* (wykresy, diagramy) do wszelkiego rodzaju publikacji, por. rozdz. 12.
— W repozytorium CRAN (*Comprehensive R Archive Network*) udostępnionych jest prawie 10 000 pakietów rozszerzających możliwości bazowego R. Zostały one stworzone przez dokładnie takich pasjonatów tego środowiska, jak my. Uniksowa filozofia tego środowiska objawia się w dobrej współpracy z innymi aplikacjami (np. programem do składu publikacji LaTeX, parserami języka Markdown) – co jeszcze lepiej pozwala nam korzystać z jego możliwości.
— R ma obszerną, łatwo dostępną *dokumentację*.

Jeśli chodzi o zastosowania, a zwłaszcza ukierunkowanie na tzw. obliczenia naukowe, najbardziej podobne do R jest środowisko Matlab oraz język Python wraz z rodziną pakietów SciPy, SciKits i Pandas, por. [29].

1.2. Organizacja pracy w R i RStudio

R jest dostępny m.in. na platformy Windows, Linux, Solaris i OS X. Jego wersję instalacyjną można pobrać ze strony www.r-project.org/. Otwieramy zakładkę CRAN,

wybieramy swój ulubiony (np. *0-Cloud*) serwer „lustrzany" (od ang. *mirror*)[2], a następnie klikamy *Download for <platforma>*. Warto w tym miejscu zauważyć, że wiele dystrybucji systemu Linux umożliwia zainstalowanie R przez właściwe sobie narzędzie (np. dnf dla systemu Fedora i Red Hat lub apt-get dla Ubuntu i Debian) – w takim przypadku będziemy mieli automatyczny dostęp do najnowszych aktualizacji zarówno samego środowiska, jak i towarzyszących mu pakietów dodatkowych.

Użytkownicy R dla Windows mogą uruchomić bardzo prosty interfejs użytkownika, klikając odpowiednią ikonę w menu *Start*. Jako że w następnym podrozdziale przejdziemy do omawiania o wiele wygodniejszego środowiska programistycznego, jakim jest RStudio, nie będziemy w tym miejscu zagłębiać się w obsługę tzw. R GUI.

R może pracować w dwóch trybach:
— w trybie interaktywnym, gdzie po każdym wydanym poleceniu otrzymujemy informację zwrotną o przebiegu jego wykonania;
— w trybie wsadowym (ang. *batch mode*), w którym zlecamy środowisku R uruchomienie danego pliku źródłowego (skryptu), czyli pliku tekstowego najczęściej o rozszerzeniu `.R`, zawierającego kolejne polecenia języka R przeznaczone do wykonania.

W codziennej pracy używa się swoistej mieszanki tych dwóch trybów, pracując raz z pojedynczymi poleceniami, a kiedy indziej z całymi zbiorami plików źródłowych (skryptów). Zobaczymy, że takie podejście będzie dla nas lada moment najbardziej naturalne.

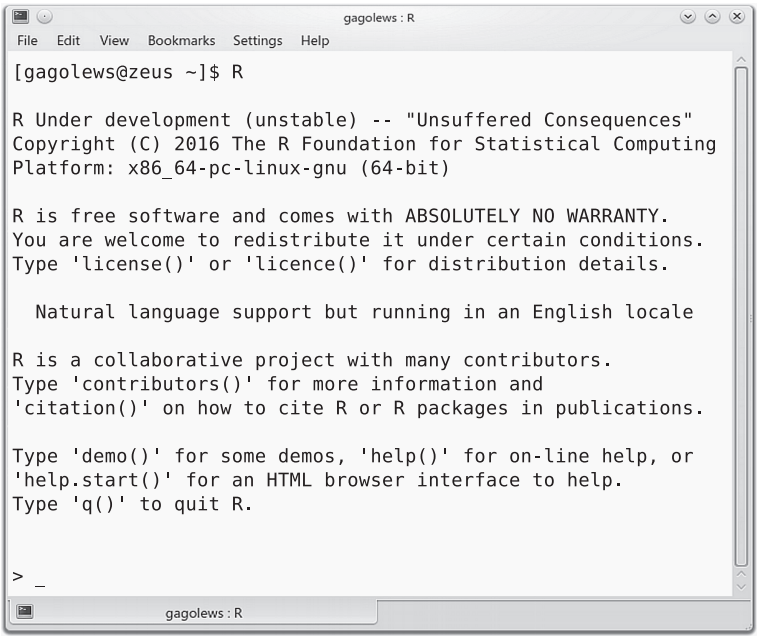
CIEKAWOSTKA

Systemy operacyjne z rodziny UNIX, a więc m.in. Solaris, OS X oraz różne dystrybucje Linuxa, są doskonale przystosowane do tworzenia oprogramowania (także w języku R). Zachęcamy więc użytkowników systemu Windows do wypróbowania jakiejś dystrybucji Linuxa – szczególnie Ubuntu i Mint są uznawane za przyjazne dla początkujących. Zainstalować ją można łatwo na maszynie wirtualnej (choćby przy użyciu Oracle VM Virtual Box), dzięki czemu nie trzeba rekonfigurować ustawień komputera – pracujemy wówczas w nowym systemie po prostu w „dodatkowym okienku".

1.2.1. Konsola R

Po uruchomieniu konsoli R (udostępnianej także przez R GUI pod Windows) zostaniemy powitani stosownymi komunikatami, por. rys. 1.1. Możemy teraz np. wyświetlić informację o zainstalowanej na naszym komputerze wersji tego środowiska, wpisując:

[2]Serwery „lustrzane" udostępniają wierne kopie zasobów projektu R. Utrzymywane są bezpłatnie, z dobrej woli ich właścicieli, przez różne instytucje naukowe i komercyjne.



Rysunek 1.1. Konsola R pod systemem Linux zaraz po uruchomieniu

```
> R.version.string
[1] "R Under development (unstable) (2016-04-14 r70486)"
> getRversion()
[1] '3.4.0'
```

Widzimy, że w niniejszym opracowaniu korzystamy z deweloperskiej wersji R, która w przyszłości zostanie oznaczona numerem 3.4.0.

Aby zakończyć bieżącą sesję, wpisujemy:

```
> q()
```

i na pytanie, czy zapisać aktualny obraz przestrzeni roboczej, odpowiadamy *Nie*.

1.2.2. Program RStudio

Bezpłatny, otwarty program RStudio Desktop można pobrać ze strony internetowej www.rstudio.com/products/rstudio/. Działa on w systemach Windows, Linux i OS X. Aplikacja ta to tzw. zintegrowane środowisko programistyczne (IDE, *integrated development environment*) dla istniejącej już w systemie instalacji R.

RStudio znakomicie ułatwia pracę ze środowiskiem R dzięki m.in.:

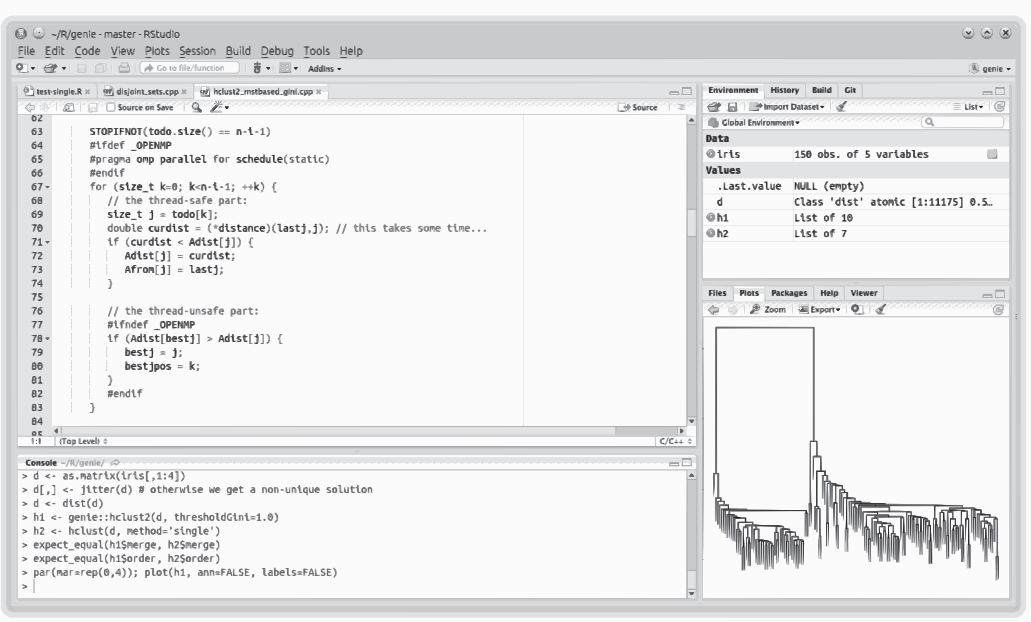
— licznym rozszerzeniom możliwości konsoli;

— wygodnemu zarządzaniu plikami źródłowymi (także w różnych kodowaniach i z automatycznym kolorowaniem składni nie tylko języka R, ale też Markdown i C++) oraz całymi projektami;
— zintegrowanemu systemowi pomocy i narzędziom wspomagającym zarządzanie generowanymi plikami graficznymi;
— obsłudze systemów kontroli wersji oprogramowania typu Git.

Co ważne, program ten tworzą osoby żywo zaangażowane w tworzenie pakietów R oraz szkolenie jego użytkowników – jest on więc dobrze dostosowany zarówno do potrzeb osób początkujących, jak i profesjonalistów.

Rysunek 1.2 przedstawia wygląd uruchomionego programu z domyślnymi ustawieniami. Okno RStudio składa się z czterech części, domyślnie:

1) edytora otwartych plików/skryptów i podglądu własności obiektów (okienko po lewej u góry);
2) znanej już nam konsoli R rozbudowanej o bardziej zaawansowane opcje edycyjne i kolorowanie składni języka (po lewej u dołu);
3) listy obiektów w sesji roboczej i historii poleceń (po prawej u góry);
4) prostego menedżera plików, podglądu rysunków, wykazu dostępnych pakietów R i przeglądarki dokumentacji (po prawej u dołu).



Rysunek 1.2. Program RStudio

Pracę w programie RStudio można organizować przy użyciu tzw. *przestrzeni roboczych* (*workspaces*), które przechowują wszystkie utworzone przez nas obiekty i całą historię wykonanych do tej pory poleceń, por. rozdz. 16. Tę funkcjonalność udostępnia

także „zwykły” R. Ponadto możemy w RStudio także tworzyć tzw. *projekty*, w których dodatkowo zapisujemy informacje m.in. o ostatnio edytowanych przez nas plikach źródłowych (skryptach) czy rysunkach. Gdy poznamy środowisko R nieco lepiej, będziemy często z nich korzystać, stosując zasadę *jedno oprogramowywane „duże” zagadnienie – jeden oddzielny projekt*. Ogólnie rzecz biorąc, przyjmuje się zasadę, że na potrzeby każdego projektu tworzymy oddzielny folder na dysku.

CIEKAWOSTKA

RStudio dostępne jest także w wersji Server. Dzięki niemu można udostępniać innym osobom możliwość przeprowadzania obliczeń w środowisku R z każdego miejsca na świecie przez przeglądarkę internetową. RStudio Server zapewnia interfejs działający i wyglądający dokładnie tak samo jak RStudio Desktop.

1.2.3. Pierwsze kroki w trybie interaktywnym

Prastara świecka tradycja głosi, że każdy kurs programowania należy zacząć od przywitania szerokiego grona odbiorców stosownym komunikatem. Powstrzymując się od przekory, wywołajmy więc następujące polecenie na konsoli. Wprowadzamy je po tzw. *znaku zachęty* (od ang. *command prompt*), tj. „>”.

```
> cat("W tak pięknych okolicznościach przyrody...\n")
```

Otrzymujemy następujący komunikat:

```
W tak pięknych okolicznościach przyrody...
```

Zauważmy, że „\n” oznacza tutaj znak nowego wiersza, tj. znak sterujący, który nakazuje przesunięcie do kolejnego wiersza aktualnej pozycji, na której jest wypisywany tekst. Co więcej, zauważmy, że napis do wydrukowania został ujęty w cudzysłów.

WAŻNE

Osoby mające choć minimalne doświadczenie z językami programowania takimi jak C, C++ czy Java od razu zauważą, że w R nie jest potrzebny żmudny czasem proces kompilacji programu. Nie ma także konieczności umieszczenia wykonywanego kodu zawsze w oddzielnych plikach źródłowych. Wydawane polecenia są interpretowane przez R w miejscu i natychmiast wykonywane.

Innymi słowy, R jest językiem interpretowanym. Ponadto właśnie skorzystaliśmy z tzw. trybu interaktywnego (pytanie → odpowiedź).

Od samego początku powinniśmy zacząć nabierać dobrych nawyków opisywania naszego kodu. Do tworzenia *komentarzy* używamy znaku „#” – każdy występujący po nim tekst, aż do końca wiersza, będzie przez interpreter ignorowany.

```
> cat("...i niepowtarzalnej...\n") # wypisuje cytat z filmu „Rejs" na konsoli
...i niepowtarzalnej...
```

Na marginesie, w wielu językach programowania konieczne jest stawianie specjalnego znaku na końcu każdego wyrażenia. Zapominanie o nim jest częstą przyczyną błędów kompilacji. W R, tak samo jak w językach C lub Java, służyć może do tego celu średnik. Jednakże średniki są w naszym przypadku potrzebne tylko wtedy, gdy chcemy z jakichś powodów zapisać wiele poleceń w jednym wierszu. W innych przypadkach to po prostu znak nowego wiersza powiadamia interpreter o końcu polecenia.

```
> # tutaj średnik jest konieczny (dwa wyrażenia w jednym wierszu):
> cat("Pani pozwoli... \n"); cat("i Pan również... \n")
Pani pozwoli...
i Pan również...
> # tutaj średnik jest zbędny (pomijanie go jest dobrą praktyką):
> cat("że skoczę po swoją żonę.\n");
że skoczę po swoją żonę.
```

Sprawdźmy, co się stanie, gdy „rozbijemy" pojedyncze polecenie na dwa wiersze. Wprowadźmy na konsoli wywołanie funkcji „`cat()`" raz jeszcze, tym razem jednak wciskając klawisz ⟨ENTER⟩ bezpośrednio po nawiasie otwierającym, „(":

```
> cat(                                                    # nowy wiersz
+ "I kto za to płaci? Pani płaci, Pan płaci.. Społeczeństwo.\n")
```

Znak zachęty „>" zmienił się na „+". Oznacza to, że R oczekuje od nas, że dokończymy wprowadzanie wyrażenia w kolejnym wierszu.

WAŻNE

Znak zachęty „+" czasem będzie pojawiał się podczas codziennej pracy z R. Bądźmy wyczuleni na jego obecność, gdyż może on wskazywać miejsce popełnienia przez nas błędu składniowego (np. niedomknięcia nawiasu czy cudzysłowu). W takiej sytuacji wciśnięcie klawisza ⟨ESC⟩ spowoduje anulowanie aktualnie wprowadzanego polecenia.

Gdy popracujemy z R nieco dłużej, zdamy sobie sprawę, że używanie tylko trybu interaktywnego, tj. wydawanie poleceń jedynie na konsoli, nie jest zbyt efektywne. W takim podejściu trudno jest m.in. wracać do uprzednio wywołanego kodu czy uruchamiać dłuższe programy. Z tego powodu omówimy za chwilę sposób zarządzania plikami źródłowymi w RStudio.

Możliwość wprowadzania wyrażeń bezpośrednio na konsoli będzie się jednak co i raz przydawać, np. do sprawdzania, czy przebieg działania naszych programów jest prawidłowy, bądź wtedy, gdy będziemy chcieli użyć R jako rozbudowanego kalkulatora.

WAŻNE

Od tej pory w niniejszym opracowaniu będziemy pomijać znaki zachęty („>” oraz „+”). Ponadto wszelkie wypisywane przez R komunikaty będziemy poprzedzać dwoma krzyżykami. Dzięki temu zwiększymy szansę uniknięcia przypadkowych błędów przy przepisywaniu fragmentów kodu.

1.2.4. Edytor skryptów

Utwórzmy w RStudio nowy plik źródłowy: wybieramy z menu *File* → *New File* → *R Script* bądź wciskamy kombinację klawiszy ⟨CTRL+SHIFT+n⟩. Skrypt ten warto od razu zapisać na dysku, np. pod nazwą `test.R` (*File* → *Save As* albo ⟨CTRL+s⟩).

WAŻNE

Warto wyrobić sobie nawyk częstego zapisywania edytowanych plików, by uniknąć przykrych niespodzianek spowodowanych np. zawieszeniem się komputera. Najczęściej do tego celu będziemy używać kombinacji klawiszy ⟨CTRL+s⟩.

Wprowadźmy w edytorze następujący kod źródłowy (zachowując formatowanie):

```
# mój pierwszy skrypt R
cat("Ja jestem umysł ścisły.\n ")
cat("Mnie się podobają melodie, ",
"które już raz słyszałem.\n")
CAT(rejs)
# koniec
```

Wielką zaletą RStudio jest to, że kod z edytora można bardzo łatwo przekazywać konsoli R. Jeśli przesuniemy karetkę (znak oznaczający aktualne miejsce, w którym wprowadzamy tekst) na pierwsze wywołanie funkcji `cat()` i wciśniemy ⟨CTRL+ENTER⟩, to R dokona natychmiast ewaluacji tego wyrażenia.

ZADANIE 1.1. Wciśnij powyższą kombinację klawiszy, gdy kursor jest ustawiony w wierszu „`cat("Mnie się podobają melodie, "`,”. Zobacz, co pojawiło się na konsoli, i pomyśl, co trzeba zrobić dalej.

Kombinacja klawiszy ⟨CTRL+ENTER⟩ zadziała także, jeśli zaznaczymy myszą jakiś fragment tekstu w edytorze. Pamiętajmy jednak, by wyrażenia bądź podwyrażenia, które chcemy wykonać na konsoli, zaznaczać w całości. Na przykład, możemy dokonać ewaluacji podwyrażenia „`"R"`” w wyrażeniu „`cat("R")`”. Przesłanie na konsolę już jednak tylko „`t("R`” spowoduje wystąpienie błędu.

ZADANIE 1.2. Zobacz, co się stanie, gdy wciśniesz ⟨CTRL+SHIFT+s⟩, a co gdy ⟨CTRL+SHIFT+ENTER⟩. Czy wszystko działa prawidłowo? Popraw samodzielnie błędy składniowe.

W edytorze skryptów działają rzecz jasna także standardowe, systemowe *skróty klawiszowe*, z którymi z pewnością spotkaliśmy się w trakcie codziennej pracy z komputerem. Wśród nich znajdują się m.in. polecenia służące do:

- obsługi schowka: ⟨CTRL+c⟩ – kopiuj, ⟨CTRL+x⟩ – wytnij, czyli kopiuj i usuń, oraz ⟨CTRL+v⟩ – wklej;
- zmiany pozycji karetki: ⟨↑⟩, ⟨↓⟩, ⟨←⟩, ⟨→⟩, ⟨PAGE DOWN⟩ – w dół o jedną „stronę” ⟨PAGE UP⟩ – w górę o jedną „stronę”, ⟨HOME⟩ – początek wiersza, ⟨END⟩ – koniec wiersza, ⟨CTRL+HOME⟩ – początek pliku, ⟨CTRL+END⟩ – koniec pliku;
- zaznaczania fragmentu tekstu bez użycia myszy: ⟨SHIFT+↑⟩, ⟨SHIFT+↓⟩ itd.;
- włączania i wyłącznia trybu zastępowania (ang. *overwrite/insert mode*) – klawisz ⟨INSERT⟩;
- wyszukiwania bądź zastępowania tekstu: ⟨CTRL+f⟩.

Powyższe kombinacje klawiszy działają także na konsoli w RStudio.

Oprócz tego w edytorze skryptów mamy możliwość:

- umieszczania aktualnego wiersza albo całego zaznaczonego bloku tekstu w komentarzu lub ponownego umieszczania go w tekście programu – ⟨CTRL+SHIFT+c⟩;
- wyróżniania całego bloku tekstu wcięciem – ⟨TAB⟩ i ⟨SHIFT+TAB⟩;
- zamykania aktualnie otwartego okna edycji – ⟨CTRL+w⟩.

Warto także wiedzieć, że możemy przenosić karetkę między edytorem a konsolą przy użyciu kombinacji klawiszy ⟨CTRL+1⟩ oraz ⟨CTRL+2⟩. Pamiętajmy, że klawiatura jest naszym „przyjacielem” i że większość czynności możemy o wiele szybciej zrealizować bez dotykania ręką myszy. Więcej informacji na temat skrótów klawiszowych znajdziemy na stronie internetowej projektu RStudio.

Do pewnych technicznych aspektów organizacji pracy w środowisku RStudio będziemy od czasu do czasu powracać. Jednakże od tej pory zakładamy, że każdy z nas potrafi wprowadzać i wykonywać polecenia R oraz poprawiać ewentualnie występujące błędy.

1.2.5. System pomocy

Jak wspomnieliśmy, środowisko R ma rozbudowany *system pomocy*. Rozważmy np. funkcję `nchar()`, która służy do zliczania liczby znaków w danym napisie:

```
nchar("Ma Pan bilet?")
## [1] 13
```

Aby otworzyć podręcznik R (ang. *R manual*) na stronie dotyczącej tej funkcji, piszemy:

```
?nchar
```

Alternatywnie:

```
?"nchar"
help("nchar")
```

Na marginesie, zapis z cudzysłowem jest przydatny, gdy chcemy uzyskać pomoc m.in. na temat operatorów; por. `?"+"`.

Większość stron podręcznika składa się z następujących działów:

1) Ogólny opis obiektu (*Description*) – zawiera informacje do czego służy np. dana funkcja.
2) Sposób użycia (*Usage*) – zawiera m.in. listę wszystkich argumentów funkcji i ich wartości domyślnych.
3) Argumenty (*Arguments*) – określa szczegółowo znaczenie poszczególnych argumentów funkcji wymienionych w sekcji *Usage*.
4) Informacje szczegółowe (*Details*) – podaje np. sytuacje, w których można użyć danej funkcji, szczegóły techniczne, krytykę stosowanych algorytmów, informacje na temat wydajności i złożoności obliczeniowej.
5) Wartość zwracana (*Value*) – dokładnie specyfikuje postać obiektu, który powstaje w rezultacie działania danej funkcji.
6) Bibliografia (*References*) – zawiera wykaz książek i artykułów, w których można znaleźć opis użytych algorytmów.
7) Odnośniki (*See Also*) do innych funkcji o podobnym działaniu.
8) Przykłady użycia (*Examples*) – podaje kod w R do samodzielnego przestudiowania i uruchomienia.

Można też uzyskać pomoc „na tematy ogólne", np.:

```
?Math
```

Podręcznik R możemy przeszukiwać przy użyciu poleceń:

```
??character
help.search("character") # równoważnie
```

WAŻNE

Jak najczęściej wyszukujmy samodzielnie interesujących nas informacji na temat obiektów środowiska R w podręczniku lub internecie. Dzięki temu pogłębimy naszą wiedzę i nabierzemy wprawy w znajdowaniu alternatywnych sposobów implementacji rozwiązywanych przez nas zagadnień. Polecamy korzystanie m.in. z forum stackoverflow.com/.

Na koniec zwróćmy uwagę, że w programie RStudio został zaimplementowany mechanizm podpowiedzi i uzupełniania wprowadzanych nazw obiektów. Po wpisaniu „ca" i naciśnięciu klawisza ⟨TAB⟩ lub kombinacji klawiszy ⟨CTRL+SPACJA⟩ ujrzymy listę obiektów, których identyfikatory zaczynają się od właśnie tego napisu.

TYPY ATOMOWE: WEKTORY I `NULL` 2

2.1. Klasyfikacja typów obiektów w języku R

Każdy język programowania można postrzegać jako narzędzie służące do instruowania komputera, w jaki sposób należy przekształcić dane wejściowe tak, by wygenerować interesujące nas dane wyjściowe. Okazuje się, że bardzo często w przetwarzanych „jednostkach informacji" da się znaleźć na poziomie ogólnym pewne podobieństwa. Na przykład, w pewnym fragmencie programu pewien obiekt może być reprezentowany w postaci skończonego ciągu liczb naturalnych oraz napisu. To, co w matematyce znamy pod pojęciem zbioru lub dziedziny, w języku R będziemy określać mianem *typu danych*.

Typy danych w R możemy sklasyfikować w następujący sposób (por. też tab. 16.5):

1) *typy podstawowe*:
   a) typy atomowe (ang. *atomic*):
      — wektor wartości logicznych (typ `logical`, p. 2.2.1);
      — wektor bajtów (typ `raw`, s. 21);
      — wektor wartości całkowitych (typ `integer`, p. 2.2.4);
      — wektor wartości rzeczywistych (typ `double`, p. 2.2.2);
      — wektor wartości zespolonych (typ `complex`, s. 22);
      — wektor napisów (typ `character`, p. 2.2.3);
      — typ pusty (typ `NULL`, podrozdz. 2.5);
   b) typy o strukturze rekurencyjnej (ang. *recursive*):
      — lista (wektor uogólniony; typ `list`, rozdz. 4);
      — funkcja (typ `closure` lub `function`, rozdz. 5);
      — środowisko (typ `environment`, rozdz. 17);
   c) typy reprezentujące nieobliczone wyrażenia języka R (rozdz. 18):
      — nazwa (symbol; typ `name`);
      — wywołanie (typ `call`);
      — ciąg wyrażeń (typ `expression`)[1];
2) *typy złożone* (ang. *compound types*), które reprezentowane są przy użyciu obiektów typów podstawowych, m.in.:

[1]Typy `call` i `expression` można także zaklasyfikować do grupy typów rekurencyjnych; por. rozdz. 18.

— macierz i tablica (klasa `matrix` oraz `array`, podrozdz. 7.1);
— szereg czasowy (klasa `ts`, podrozdz. 7.2)
— czynnik (klasa `factor`, podrozdz. 7.3);
— ramka danych (klasa `data.frame`, podrozdz. 7.4);
— formuła (klasa `formula`, podrozdz. 18.4).

W tym rozdziale poznamy sposób tworzenia obiektów typu atomowego, czyli takich, które mają w pewnym sensie jednorodną strukturę. Dowiemy się także, w jaki sposób możemy tworzyć obiekty nazwane oraz w jaki sposób R oznacza tzw. wartości specjalne, np. braki danych. Dodatkowo, w rozdz. 3 zaprezentujemy najważniejsze operacje, które można wykonywać na obiektach typu atomowego.

Następnie rozszerzymy naszą wiedzę m.in. o pozostałe typy podstawowe oraz sposoby tworzenia nowych funkcji (choć funkcji wbudowanych będziemy używać już za chwilę). Na tych solidnych fundamentach będziemy budować naszą umiejętność zastosowania R w różnych problemach praktycznych. Tym samym bardzo szybko okaże się, że niemała liczba zagadnień omawianych w dalszej części książki będzie „tylko" twórczą kombinacją treści przedstawionych w rozdz. 2–9.

2.2. Wektory atomowe

Zacznijmy od omówienia trzech najczęściej używanych typów atomowych: wektorów logicznych, liczbowych i napisów.

2.2.1. Wektory wartości logicznych

W R zdefiniowane zostały następujące *dwie stałe logiczne*:

```
TRUE   # prawda
## [1] TRUE
FALSE  # fałsz
## [1] FALSE
```

WAŻNE

R jest językiem, w którym wielkość liter ma znaczenie (ang. *case-sensitive*). Odwołując się powyżej np. do „`false`" bądź „`True`", nie uzyskamy spodziewanego rezultatu.

Przyjrzyjmy się dwóm funkcjom służącym do tworzenia wektorów. Działają one w bardzo ogólny i elastyczny sposób – możemy przy ich użyciu tworzyć także nie tylko wektory logiczne, ale i wektory innych typów.

Tworzenie wektorów przez złączanie. Funkcji `c()` (od ang. ***c**ombine*, czyli złącz) używamy do tworzenia wektora (ciągu) składającego się z zadanych wartości logicznych występujących w określonej kolejności.

```
c(TRUE, FALSE, FALSE, TRUE)
## [1]  TRUE FALSE FALSE  TRUE
c(c(TRUE, FALSE), c(FALSE, TRUE))    # złączenie dwóch wektorów jest wektorem
## [1]  TRUE FALSE FALSE  TRUE
```

Długość danego wektora możemy poznać, wywołując funkcję `length()`.

```
length(c(TRUE, TRUE, FALSE))         # ile wynosi długość tego wektora?
## [1] 3
```

Co ciekawe, *w R nie operujemy na typowych wartościach skalarnych*. Nawet pojedyncza wartość logiczna jest przechowywana w postaci wektora o długości jeden. Znakomicie ułatwia to pracę: dzięki temu, np. w pisanych przez nas funkcjach, nie musimy rozpatrywać oddzielnych przypadków dla wartości skalarnej oraz wektora.

```
length(FALSE)
## [1] 1
c(FALSE)                             # to samo, co po prostu „FALSE”
## [1] FALSE
```

Tworzenie wektorów przez replikację. Wektory możemy utworzyć także przy użyciu funkcji `rep()` (od ang. ***rep**licate*), która powtarza (replikuje) ciąg zadanych wartości.

```
rep(TRUE, 3)
## [1] TRUE TRUE TRUE
```

Powyższe wywołanie dało nam w wyniku wektor, w którym wartość `TRUE` została powtórzona trzy razy.

ZADANIE 2.1. Zajrzyj teraz do dokumentacji funkcji `rep()`, wywołując `?rep` albo wpisując do swojej ulubionej wyszukiwarki internetowej frazę „`R Documentation rep`”. Stąd dowiesz się, że funkcja ta rozpoznaje argumenty o nazwie *`times`*, *`each`* oraz *`length.out`*.

W powyższym przykładzie to właśnie argument *`times`* jest używany domyślnie. W związku z tym następujące dwa wyrażenia są równoważne.

```
rep(c(TRUE, FALSE), 3)        # replikuj wektor
## [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE
rep(c(TRUE, FALSE), times=3) # to samo
## [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE
```

Aby nadać wartość argumentowi *each*, musimy podać jego nazwę w sposób jawny.

```
rep(c(TRUE, FALSE), each=3) # powtórz każdy element
## [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE
rep(c(TRUE, FALSE), , , 3)  # each jest 4. argumentem (mało czytelne)
## [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE
```

Widzimy, że zachowanie funkcji **rep()** się zmieniło. Zamiast powtórzenia całego wektora z zastosowaniem swego rodzaju „zawijania", każda wartość została kolejno powtórzona.

Określenie wartości argumentu *length.out* powoduje utworzenie wektora o zadanej długości wyjściowej.

```
rep(c(TRUE, FALSE), length.out=3)
## [1]  TRUE FALSE  TRUE
rep(c(TRUE, FALSE), , 3) # jest to trzeci argument (znów mało czytelne)
## [1]  TRUE FALSE  TRUE
```

Odnotujmy, że domyślnie stosowana zasada powtarzania podanych elementów wektora jest tutaj podobna do tej, którą udostępnia argument *times*. Możemy jednak to zmienić, używając różnych kombinacji omawianych trzech argumentów:

```
rep(c(TRUE, FALSE), times=2, each=2)
## [1]  TRUE  TRUE FALSE FALSE  TRUE  TRUE FALSE FALSE
rep(c(TRUE, FALSE), length.out=8, each=3)
## [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE  TRUE
```

WAŻNE

Okazuje się, że nie trzeba odwoływać się do pełnych nazw parametrów przy wywołaniu funkcji. R sam uzupełni brakującą część nazwy, jeśli tylko jej rozwinięcie jest jednoznaczne. Wobec tego poniższe wywołania dają dokładnie ten sam wynik.

```
rep(c(TRUE, FALSE), length.out=3)
## [1]  TRUE FALSE  TRUE
rep(c(TRUE, FALSE), length=3) # to samo
## [1]  TRUE FALSE  TRUE
rep(len=3, c(TRUE, FALSE))     # zmieniona kolejność, ale dobrze określone
## [1]  TRUE FALSE  TRUE
rep(len=3, x=c(TRUE, FALSE))  # pierwszy parametr nazywa się x, zob. ?rep
## [1]  TRUE FALSE  TRUE
```

Taki mechanizm nazywamy *częściowym dopasowaniem* (ang. *partial matching*) bądź *autouzupełnianiem* nazw argumentów. Choć jest on wygodny podczas codziennej pracy,

programiści bardziej złożonych aplikacji nie powinni z niego korzystać. Nigdy nie wiadomo, czy w przyszłości twórcy R nie rozszerzą implementacji używanych przez nas funkcji wbudowanych o dodatkowe parametry, które będą w konflikcie z naszymi oczekiwaniami.

2.2.2. Wektory liczbowe

Każdy ciąg cyfr jest uznawany za *stałą liczbową*. W przypadku chęci podania liczby o niezerowej części ułamkowej część dziesiętną od ułamkowej powinniśmy oddzielić kropką.

```
1                       # to jest wektor liczbowy o długości jeden, zob. length(1)
## [1] 1
c(1, -2, +3, 4., -.5) # różne „chwyty" dozwolone
## [1]  1.0 -2.0  3.0  4.0 -0.5
rep(3.14, 2)            # rep() też działa
## [1] 3.14 3.14
```

Przy wprowadzaniu liczb możemy także korzystać z tzw. *notacji naukowej*, w której znak „e" (od *exponent*) służy do „przesuwania" kropki dziesiętnej w lewo lub w prawo. Na przykład, `1.2e-2` oznacza $1{,}2 \cdot 10^{-2}$.

```
1.2e-2
## [1] 0.012
1e-16                   # bardzo mała liczba, „prawie" zero
## [1] 1e-16
```

Generowanie ciągów arytmetycznych. Oprócz funkcji `c()` i `rep()` do tworzenia wektorów liczbowych możemy wykorzystać także operator „`:`" (dwukropek). Generuje on ciągi arytmetyczne o różnicach równych zawsze 1 bądź -1.

```
c(-2:2, 5:1)
##  [1] -2 -1  0  1  2  5  4  3  2  1
1.1:5.5                 # do 5.1 (dlaczego?)
## [1] 1.1 2.1 3.1 4.1 5.1
```

Ciągi arytmetyczne o dowolnych różnicach generujemy przy użyciu funkcji `seq()` (od ang. ***seq**uence*).

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

ZADANIE 2.2. Przeczytaj dokumentację funkcji `seq()`, wywołując `?seq`. Zwróć szczególną uwagę na sekcję *Arguments*.

Okazuje się, że trzeci argument tej funkcji nazywa się *by* i odpowiada właśnie za wielkość różnicy. Możemy go jednak pominąć i zamiast niego podać *length.out*. Podobnie jak w przypadku funkcji `rep()` odpowiada on za tworzenie wektora o zadanej długości.

```
seq(0, 1, length.out=5)
## [1] 0.00 0.25 0.50 0.75 1.00
```

Wielkość różnicy ciągu zostaje tutaj wyznaczona automatycznie przy użyciu wzoru $by = (to - from)/\max\{length.out - 1, 1\}$.

2.2.3. Wektory napisów

Z pojedynczymi *napisami*, tj. ciągami dowolnych znaków drukowanych, mieliśmy już do czynienia w rozdziale wprowadzającym. Do ich tworzenia mogą służyć cudzysłów bądź apostrofy.

```
"Zaniemogło biedactwo."          # równoważnie: 'Zaniemogło biedactwo.'
## [1] "Zaniemogło biedactwo."
length("Zaniemogło biedactwo.") # jest to wektor napisów o długości 1
## [1] 1
```

Co ciekawe, mimo że napis sam w sobie jest już ciągiem znaków, w R operujemy na wektorach *całych* napisów, a więc ciągach ciągów znaków.

```
c("kolka", "wątroba", "śledziona", "noga")
## [1] "kolka"     "wątroba"   "śledziona" "noga"
length(c("kolka", "wątroba", "śledziona", "noga"))
## [1] 4
```

Znaki specjalne. Definiując napis, niektórych znaków nie da się wprowadzić wprost – dotyczy to na przykład cudzysłowu czy apostrofu. W związku z tym uznano, że pewne ciągi znaków – zawsze poprzedzone odwróconym ukośnikiem (ang. *backslash*) – będą miały specjalne znaczenie. W szczególności, aby wprowadzić po prostu odwrócony ukośnik, należy poprzedzić go…tym samym znakiem. Wykaz najczęściej spotykanych *znaków specjalnych* (ang. *escape characters*) znajduje się w tab. 2.1. Oto kilka przykładów:

```
"Napisy twórz \"tak\" albo \'tak\'" # tutaj niepotrzebny ukośnik przed apostrofem
## [1] "Napisy twórz \"tak\" albo 'tak'"
```

```
print("Napisy twórz \"tak\" albo 'tak'") # równoważnie
## [1] "Napisy twórz \"tak\" albo 'tak'"
cat("Napisy twórz \"tak\" albo 'tak'")
## Napisy twórz "tak" albo 'tak'
cat('Napisy twórz "tak" albo \'tak\'')
## Napisy twórz "tak" albo 'tak'
cat("Aby wprowadzić \\, użyj \\\\.")
## Aby wprowadzić \, użyj \\.
```

Dalej, wywołując `cat("aaac\rbbb\n")`, otrzymamy na konsoli ciąg znaków „bbbc", zaś stosując `cat("123\b45\n")` – „1245". Zwróćmy uwagę na to, że w przeciwieństwie do funkcji `cat()`, `print()` nie odkrywa prawdziwego znaczenia znaków specjalnych – wypisuje na konsoli napisy tak, jak je wprowadzamy ręcznie.

Tabela 2.1. Wybrane znaki specjalne; zob. więcej, wywołując `?Quotes`

Znak	Znaczenie
`\n`	nowy wiersz
`\r`	powrót karetki do początku wiersza
`\t`	tabulator
`\b`	usunięcie znaku poprzedzającego (*backspace*)
`\\`	odwrócony ukośnik (*backslash*)
`\'`	apostrof (dot. napisów `'...'`)
`\"`	cudzysłów (dot. napisów `"..."`)

2.2.4. Pozostałe typy wektorów atomowych i ich hierarchia

Typ podstawowy *każdego* obiektu R możemy poznać, wywołując funkcję `typeof()`: Omówione rodzaje wektorów atomowych są identyfikowane w następujący sposób:

```
c(typeof(TRUE), typeof(7), typeof("Ma Pan bilet?"))
## [1] "logical"   "double"    "character"
```

Podobną funkcją jest `mode()`. Wynik przez nią zwracany często pokrywa się z tym otrzymywanym przez `typeof()`. W niektórych jednak przypadkach jest on bardziej „przyjazny dla użytkownika".

```
c(mode(TRUE), mode(7), mode("Ma Pan bilet?"))
## [1] "logical"   "numeric"   "character"
```

Dla kompletności opiszemy teraz trzy pozostałe typy wektorów atomowych – znacznie rzadziej używane w praktyce.

Wektory wartości całkowitych. Formalnie, to, co ogólnie nazywamy „wektorem liczbowym", w pamięci komputera może być reprezentowane przy użyciu dwóch następujących typów podstawowych:

i) całkowite (`integer`);
ii) rzeczywiste (zmiennopozycyjne, `double`).

} liczbowe (`numeric`)

Mimo rozróżnienia typów całkowitych i rzeczywistych w R (które doskonale znają programiści np. języków C lub Python), wprowadzane stałe liczbowe i tym samym wektory tworzone przez wywołanie funkcji `c()`, `rep()` lub `seq()` są traktowane najczęściej jako zmienne typu rzeczywistego – nawet jeśli części ułamkowe tych liczb są równe 0.

```
c(typeof(1), mode(1)) # liczba całkowita, ale typ rzeczywisty
## [1] "double"  "numeric"
c(typeof(1.1), mode(1.1))
## [1] "double"  "numeric"
```

Takie zachowanie R jest wygodne z punktu widzenia jego najbardziej popularnych zastosowań, czyli obliczeń naukowych i analizy danych.

Jeśli jednak z pewnych powodów chcielibyśmy wprowadzić wartość typu całkowitego, można to zrobić, korzystając z przyrostka „L".

```
c(typeof(1L), mode(1L))
## [1] "integer" "numeric"
```

Co ciekawe, ze względów wydajnościowych operator „:" generuje czasem ciągi o typie podstawowym całkowitym – może to przyspieszać działanie np. pętli.

```
typeof(1:5)              # podobnie: seq(1, 5), ale nie seq(1, 5, 1)
## [1] "integer"
typeof(1.1:5.5)
## [1] "double"
```

CIEKAWOSTKA

Powtórzmy, przeważnie nie powinno nas interesować, czy liczba reprezentowana jest przy użyciu typu podstawowego `integer`, czy też `double`. Rozróżnienie to jest spowodowane tym, że najczęściej za obliczenia na różnych typach liczbowych są odpowiedzialne różne jednostki procesora komputera (ALU lub FPU).

Typ podstawowy `integer` stanową 32-bitowe liczby całkowite ze znakiem zapisane w systemie uzupełnień do dwóch (U2), tj. reprezentuje on liczby całkowite z zakresu $\pm$ ok. 2 miliardy.

```
.Machine$integer.max     # maksymalna wartość typu integer
## [1] 2147483647
.Machine$integer.max+1L # już nie „mieści" się w typie integer
## Warning in .Machine$integer.max + 1L: NAs produced by integer
overflow
## [1] NA
```

Z kolei `double` (64-bitowy typ zmiennopozycyjny) może reprezentować dokładnie wszystkie liczby całkowite z przedziału ok. $[-9 \cdot 10^{15}, 9 \cdot 10^{15}]$. Największą reprezentowalną liczbą całkowitą jest w tym przypadku:

```
print(2^.Machine$double.digits, digits=22)    # większa dokładność
## [1] 9007199254740992
print(2^.Machine$double.digits+1, digits=22) # identyczna wartość
## [1] 9007199254740992
```

Z praktycznego punktu widzenia bardziej korzystne jest więc reprezentowanie liczb całkowitych przy użyciu typu podstawowego `double` (kosztem niewielkiego spowolnienia obliczeń).

Zwróćmy też uwagę, że z podrozdz. 14.1 dowiemy się o ważnych w przypadku przeprowadzania obliczeń numerycznych, podstawowych własnościach i ograniczeniach arytmetyki zmiennopozycyjnej komputera.

Wektory bajtów. Wektory bajtów są reprezentowane przez typ `raw`. Można w nich przechowywać wyłącznie wartości ze zbioru {0, 1, . . . , 255}. Tego rodzaju obiekty mogą się przydać np. w przypadku przetwarzania napisów w pewnych kodowaniach bądź plików binarnych, por. rozdz. 10 i 11.

```
as.raw(c(1, 9, 10, 11, 15, 16, 100, 255))
## [1] 01 09 0a 0b 0f 10 64 ff
c(typeof(as.raw(1)), mode(as.raw(1)))
## [1] "raw" "raw"
```

Elementy wektora typu `raw` są wypisywane na konsoli w notacji szesnastkowej (heksadecymalnej).

CIEKAWOSTKA

Liczba w systemie arytmetycznym o podstawie 16 (liczba szesnastkowa) jest zapisywana przy użyciu cyfr ze zbioru {0, 1, . . . , 9, a, . . . , f}, gdzie $\mathrm{a} = 10_{10}, \mathrm{b} = 11_{10}, \ldots, \mathrm{f} = 15_{10}$ (dziesiętnie). Szesnastkowa liczba dwucyfrowa postaci x_1x_0 w systemie dziesiętnym ma wartość $x_1 \cdot 16^1_{10} + x_0 \cdot 16^0_{10} = 16_{10}x_1 + x_0$. Na przykład, $\mathrm{ca}_{16} = 16_{10} \cdot 12_{10} + 10_{10} = 202_{10}$ oraz $7f_{16} = 16_{10} \cdot 7_{10} + 15_{10} = 127_{10}$.

Liczbę y w postaci dziesiętnej ze zbioru $\{0, 1, \ldots, 255\}$ można też przedstawić w postaci dwucyfrowej szesnastkowej x_1x_0 przy użyciu przekształcenia $x_1 = y\%/\%16$ (dzielenie całkowite), $x_0 = y\%\%16$ (modulo).

Na marginesie, w razie potrzeby wprowadzenia wartości bezpośrednio w systemie szesnastkowym (heksadecymalnym) stosuje się przedrostek „0x".

```
0x21
## [1] 33
as.raw(0x21)
## [1] 21
```

Wektory wartości zespolonych. Pewna grupa odbiorców tej książki może być zainteresowana działaniami na *wartościach zespolonych*. Tak zwana jednostka urojona, oznaczana jako i, jest równa z definicji rozwiązaniu równania $i^2 = -1$. Każdą wartość ze zbioru liczb zespolonych, ozn. $\mathbb{C}$, można przedstawić w postaci algebraicznej $x + iy$, gdzie $x, y \in \mathbb{R}$. Dzięki temu elementy ze zbioru $\mathbb{C}$ możemy postrzegać jako „dwuwymiarowe" liczby rzeczywiste, z których każda składa się z części rzeczywistej x i urojonej y.

Do wprowadzania wartości urojonych korzystamy z przyrostka „i".

```
1+1.5i
## [1] 1+1.5i
c(typeof(1+1.5i), mode(1+1.5i))
## [1] "complex" "complex"
c(0i, 1i, 2i)
## [1] 0+0i 0+1i 0+2i
seq(1+1i, 3+6i, length.out=3)
## [1] 1+1.0i 2+3.5i 3+6.0i
```

Hierarchia i uzgadnianie typów. Wszystkie omówione rodzaje wektorów przechowują elementy jednego ściśle określonego typu: powiemy, że mają jednorodną, *atomową* strukturę. Rozważmy wobec tego prosty eksperyment. Co się stanie, gdy spróbujemy przy użyciu funkcji `c()` utworzyć wektor składający się z elementów należących do odmiennych dziedzin?

```
c(FALSE, 1L, 2.5, 3.0i, "cztery")  # wektor napisów
## [1] "FALSE"  "1"      "2.5"    "0+3i"   "cztery"
c(FALSE, 1L, 2.5, 3.0i)            # wektor wartości zespolonych
## [1] 0.0+0i 1.0+0i 2.5+0i 0.0+3i
c(FALSE, 1L, 2.5)                  # wektor liczb rzeczywistych
## [1] 0.0 1.0 2.5
```

```
c(FALSE, 1L)                          # wektor liczb całkowitych
## [1] 0 1
typeof(c(FALSE, 1L, 2.5))
## [1] "double"
typeof(c(FALSE, 1L))
## [1] "integer"
```

Gdy łączymy ze sobą wektory różnych rodzajów, R *uzgodni* ich typ tak, żeby informację o najbardziej „ogólnym" z obiektów dało się przechować bez znaczącej[2] straty. Taki mechanizm nazywamy *uzgadnianiem typów* bądź *koercją* (ang. *coercion*).

Z powyższego przykładu i innych podobnych eksperymentów możemy wywnioskować, że *hierarchia typów wektorów atomowych* w R wygląda następująco (w kolejności od najbardziej do najmniej szczegółowego):

1) wektor wartości logicznych (typ `logical`);
2) wektor wartości całkowitych (typ `integer`);
3) wektor wartości rzeczywistych (typ `double`);
4) wektor wartości zespolonych (typ `complex`);
5) wektor napisów (typ `character`).

Rzutowanie typów. Wszystkie wektory można także w sposób jawny *rzutować* na wektory innych typów przy użyciu funkcji `as.character()`, `as.complex()`, `as.double()` (równoważnie: `as.numeric()`), `as.integer()`, `as.raw()` oraz `as.logical()`). W takich przypadkach zezwalamy świadomie na ewentualną utratę informacji, która może nastąpić przy rzutowaniu z typu ogólniejszego do bardziej szczegółowego.

WAŻNE

Wartości logiczne `TRUE` i `FALSE` są przedstawiane liczbowo zawsze jako, odpowiednio, `1` i `0`. Lecz wartość liczbowa nierówna zeru zawsze po zrzutowaniu do logicznej daje wynik równy `TRUE`.

Kilka przykładów:

```
as.numeric(c(TRUE, FALSE))
## [1] 1 0
as.complex(c(TRUE, FALSE))                    # też 1 i 0
## [1] 1+0i 0+0i
as.character(c(TRUE, FALSE))                  # a teraz do napisów
## [1] "TRUE"  "FALSE"
as.logical(-2:2)
## [1]  TRUE  TRUE FALSE  TRUE  TRUE
```

[2] Konwersja liczb rzeczywistych na napisy może czasem spowodować ich zaokrąglenie.

```
as.logical(0+4i)                                    # nie-zero
## [1] TRUE
as.logical(c("FALSE","false","F","f","FAL","T")) # o wartości NA dalej...
## [1] FALSE FALSE FALSE    NA    NA  TRUE
```

WAŻNE

Rzutowanie liczby rzeczywistej do całkowitej następuje przez *obcięcie* jej części ułamkowej.

```
as.integer(c(1.5, 1.6, -1.5))             # obcięcie części ułamkowej
## [1]  1  1 -1
as.integer(c("1", "-1.5", "???", "1e5")) # sprytne
## [1]      1     -1     NA 100000
```

Do rzutowania typów możemy stosować także ogólniejszą funkcję `as.vector()`. Jej drugi argument, o nazwie *mode*, określa docelowy tryb (por. `?mode`) bądź typ podstawowy (por. `?typeof`) wynikowego wektora.

```
as.vector(c(TRUE, FALSE), "numeric")
## [1] 1 0
```

Testowanie typu obiektu. Istnieją także funkcje służące do sprawdzania, czy dany obiekt jest określonego typu: `is.character()`, `is.complex()`, `is.numeric()` (równoważne: `is.double()` *lub* `is.integer()`), `is.raw()` oraz `is.logical()`). Będziemy je później wykorzystywać m.in. do weryfikacji poprawności argumentów funkcji:

```
c(is.integer(1:5), is.double(1:5), is.numeric(1:5))
## [1]  TRUE FALSE  TRUE
```

Dostępne są również funkcje do weryfikacji typu obiektów na nieco bardziej ogólnym poziomie. Na przykład, przy użyciu poniższych wywołań możemy dowiedzieć się, że do tej pory mieliśmy do czynienia właśnie z *wektorami atomowymi*.

```
is.vector(c(1,2,3))  # wektor...
## [1] TRUE
is.atomic(c(1,2,3))  # ...atomowy (są też wektory uogólnione)
## [1] TRUE
```

Prealokacja wektorów. Czasem będzie zachodzić potrzeba utworzenia wektora o zadanej długości i określonym typie elementów, który dopiero później sukcesywnie bę-

dziemy wypełniać konkretnymi wartościami. Mimo że w takiej sytuacji da się zamierzony rezultat uzyskać, korzystając z funkcji `rep()`, to jednak o wiele wygodniej i szybciej jest wywołać np.:

```
logical(3)                          # tworzy wektor wartości logicznych o długości 3
## [1] FALSE FALSE FALSE
vector(mode="logical", length=3) # to samo
## [1] FALSE FALSE FALSE
vector("logical", 3)                # to samo
## [1] FALSE FALSE FALSE
integer(2)                          # albo vector("integer", 2)
## [1] 0 0
numeric(5)                          # albo double(5) itp.
## [1] 0 0 0 0 0
complex(3)
## [1] 0+0i 0+0i 0+0i
character(2)
## [1] "" ""
```

Zapamiętajmy, jaka wartość jest domyślnie nadawana elementom przez ww. funkcje: jest to fałsz, zero bądź pusty napis.

2.3. Tworzenie obiektów nazwanych

Do tej pory tworzyliśmy obiekty tylko po to, by zobaczyć je na konsoli. Jednak w codziennej pracy z R bardzo często zachodzi potrzeba zapamiętania informacji w pewnym określonym miejscu pamięci komputera, choćby po to, by za chwilę móc dokonać na niej jakiejś operacji.

Do danych przechowywanych w pamięci możemy odwoływać się przy użyciu tzw. *obiektów nazwanych*. Aby utworzyć obiekt nazwany, należy związać (ang. *bind*) pewną nazwę[3] (identyfikator) z jakąś wartością. Taką czynność możemy wykonać m.in. przy użyciu jednego z trzech dostępnych w R operatorów przypisania (ang. *assignment operators*):

```
nazwa <- wartość
wartość -> nazwa
nazwa = wartość # nie zalecamy
```

Jako że operator „=” ma również inne znaczenia (pamiętamy np., że używaliśmy go wyżej do ustalania wartości argumentów funkcji), jego stosowanie do interesujących nas tutaj celów jest niewskazane.

[3] Nazwy są wiązane w środowiskach; zob. rozdz. 17 i 18.

Nazwy syntaktyczne. Nazwa pozwala na jednoznaczne wskazanie interesującego nas obiektu w danym kontekście obliczeń. Z powodu takiej a nie innej składni języka R (por. jednak rozdz. 17 i 18) zalecane jest stosowanie pewnych ograniczeń co do postaci nadawanych nazw.

WAŻNE

Tak zwane *nazwy syntaktyczne* mogą się składać z ciągów liter, cyfr, kropek („.”) i podkreślników („_”). Nie mogą jednak rozpoczynać się od cyfry, podkreślnika oraz od cyfry poprzedzonej kropką; por. także `?make.names`.

Za niepoprawne jest również uznawane użycie tzw. *słów kluczowych* języka R. Wśród nich znajdziemy następujące identyfikatory (zob. `?Reserved`): `TRUE`, `FALSE`, `NULL`, `Inf`, `NaN`, `NA`, `NA_integer_`, `NA_real_`, `NA_complex_`, `NA_character_`, `if`, `else`, `repeat`, `while`, `function`, `for`, `in`, `next`, `break`, a także „`...`”. Poprawnymi nazwami syntaktycznymi są więc np.: `n`, `x2`, `wartosc_calki` oraz `.wynik`.

WAŻNE

Powinniśmy zwracać uwagę, by wybrane nazwy obiektów opisywały (dokumentowały) znaczenie związanych z nimi wartości. Z tego powodu dla obiektu liczbowego równego $\pi/2$, identyfikator `pi.pol` jest o wiele lepszy niż `abcdef`. O pewnych konwencjach nazewniczych możemy także przeczytać m.in. w [4].

Ponadto, choć jest to dozwolone, raczej nie powinniśmy używać polskich znaków diakrytycznych (ą, ś, ...) w identyfikatorach: tzw. reszta świata może mieć z nimi problemy.

Wartość związywana z nazwą. Operacja przypisania – oprócz nazwy – potrzebuje jeszcze drugiego elementu do pary. Związywaną *wartość* możemy uzyskać, wprowadzając np. stałą liczbową albo wywołując pewną funkcję. Ogólnie, wartość taka powstaje w wyniku ewaluacji jakiegoś wyrażenia.

CIEKAWOSTKA

W języku C i wielu innych językach kompilowanych zmienne (które najbardziej przypominają obiekty języka R związane z jakąś nazwą) należy zadeklarować przed ich pierwszym użyciem. W R wcale nie ma takiej potrzeby: jeśli chcemy związać nazwę z wartością, wystarczy po prostu użyć operatora przypisania.

Skutki użycia operatora przypisania. Utwórzmy obiekt związany z nazwą `x`, będący wektorem liczbowym złożonym z pięciu kolejnych liczb naturalnych.

```
x <- 1:5              # przypominamy: nazwa „x" nie jest tożsama z nazwą „X"
```

W tym momencie nazwa „x" powinna pojawić się w zakładce *Workspace* programu RStudio, w sekcji *Values*. Zauważmy także, że wykonanie operacji przypisania nie skutkuje automatycznie wypisaniem żadnego wyniku na konsoli. Możemy jednak takie zachowanie wymusić, ujmując całe wyrażenie w nawiasy okrągłe.

```
(x <- 1:5)
## [1] 1 2 3 4 5
```

Użycie obiektu nazwanego. Wartość związaną z nazwą „x" możemy wypisać na konsoli na różne sposoby, np.:

```
x                 # po prostu x
## [1] 1 2 3 4 5
print(x)          # jw.
## [1] 1 2 3 4 5
str(x)            # bardziej „technicznie"
##  int [1:5] 1 2 3 4 5
cat(x)            # to raczej tylko dla napisów, ale też działa
## 1 2 3 4 5
```

Oczywiście ewaluacja powyższego wyrażenia dałaby ten sam wynik, jeśli zamiast nazwy „x" podstawilibyśmy po prostu `1:5`.

Co ważne, z każdą nazwą da się związać obiekt dowolnego typu: typ nie jest ustalany raz na zawsze. Możemy stworzyć np. wektor liczbowy, by zaraz potem „podmienić" go jakimś napisem.

```
(y <- 1:3)
## [1] 1 2 3
(y <- as.complex(y))
## [1] 1+0i 2+0i 3+0i
(y <- FALSE)
## [1] FALSE
```

Zauważmy, że funkcje działają tak samo niezależnie od tego, czy jako argumenty podajemy im bezpośrednio jakieś wartości, czy też dokonamy tego pośrednio – przez nazwę obiektu. W drugim przypadku nastąpi automatyczne odczytanie wartości związanej z daną nazwą. Ze zmiennymi możemy więc robić, co nam się żywnie podoba:

```
(x <- as.complex(x))
## [1] 1+0i 2+0i 3+0i 4+0i 5+0i
(x <- c(x, x, 0))
```

```
##  [1] 1+0i 2+0i 3+0i 4+0i 5+0i 1+0i 2+0i 3+0i 4+0i 5+0i 0+0i
(x <- c("O", "A"))
## [1] "O" "A"
rep(x, each=3)
## [1] "O" "O" "O" "A" "A" "A"
ile <- 4
usmiech <- ":)"
rep(usmiech, ile)
## [1] ":)" ":)" ":)" ":)"
```

CIEKAWOSTKA

Jako że operator przypisania działa zgodnie z zasadą „od prawej do lewej” (por. s. 41), możemy zatem napisać:

```
x <- y <- 5
x <- (y <- 5)       # to samo
y <- 5; x <- y      # to samo
```

R należy do klasy języków funkcyjnych – każda wykonywana „operacja” jest de facto sprowadzana do wywołania pewnej funkcji. W powyższym przykładzie wynikiem wykonania operacji przypisania jest przypisywana wartość. Tak naprawdę wyrażenie:

```
u <- 5
```

jest interpretowane przez parser jako:

```
"<-"("u", 5)
```

co oznacza: „wywołaj funkcję `"<-"()` z argumentami `"u"` i `5`”.

WAŻNE

`TRUE` i `FALSE` to zarezerwowane słowa kluczowe języka R. Zatem ich znaczenie jest zawsze dobrze określone niezależnie od kontekstu. Warto zwrócić jednak uwagę, iż dla wygody zostały także określone ich synonimy (które już słowami kluczowymi nie są), odpowiednio, `T` oraz `F`. Odradzamy ich stosowanie, ponieważ zbyt łatwo jest „nadpisać” wartość tych obiektów. Są one bowiem tworzone automatycznie w następujący sposób:

```
T <- TRUE
F <- FALSE
```

Musimy więc być bardzo ostrożni: nic nie stoi na przeszkodzie, aby podmienić ich wartość czymkolwiek innym.

```
T <- FALSE
print(T)
## [1] FALSE
rm(T) # usuń wiązanie nazwy T, przywracając tym samym jej oryginalną wartość
T
## [1] TRUE
```

2.4. Braki danych, wartości nieskończone i nie-liczby

W analizie danych często zachodzi potrzeba oznaczenia pewnych wartości jako „nieznane" albo „niedostępne". W statystyce taką sytuację określamy mianem *braków danych* (ang. *missing data*). Teoria dotycząca radzenia sobie z brakami danych w analizie statystycznej jest wbrew pozorom dość rozbudowana i stanowi ciekawy temat badawczy, por. np. [78].

Braki danych w R. W R do reprezentacji takich przypadków służy obiekt `NA` (ang. *not available*; zob. `?NA`). Dla przypomnienia, taką wartość uzyskaliśmy już wcześniej podczas rzutowania typów. Gdy wywołaliśmy `as.integer("???")`, R odpowiedział zgodnie ze swoją najlepszą wiedzą: „nie wiem, co to jest".

Do sprawdzania, które elementy wektora nie są znane, możemy użyć funkcji `is.na()`.

```
is.na(c(1, 2, NA, 4))
## [1] FALSE FALSE  TRUE FALSE
```

Większość operacji na wartościach `NA` w wyniku da także `NA`, co jest w zasadzie zgodne z naszymi oczekiwaniami.

```
10 == NA        # porównywanie: czy 10 jest równe nie-wiadomo-czemu? Nie wiadomo...
## [1] NA
2^c(0, 1, NA, 3) # potęgowanie
## [1]  1  2 NA  8
```

Temat działań na brakach danych omówimy szerzej w rozdz. 3.

CIEKAWOSTKA

Stała `NA` jest typu logicznego. Istnieją jednak odmiany `NA` dla innych typów podstawowych (oprócz `raw`; por. wykaz zarezerwowanych słów kluczowych na s. 26). R stosuje je niejawnie zgodnie z własnymi potrzebami. Wszystko po to, by nie naruszyć zasady spójności typów elementów wektorów atomowych.

```
typeof(NA)
## [1] "logical"
NA_integer_      # brak danych jako liczba całkowita wypisywana po prostu jako NA
## [1] NA
typeof(NA_integer_)
## [1] "integer"
is.na(NA_integer_)
## [1] TRUE
```

Zauważmy także, iż:

```
is.na("NA")      # to nie to samo...
## [1] FALSE
is.na(as.logical("NA"))
## [1] TRUE
```

Inne wartości specjalnego znaczenia. `NA` należy do klasy tzw. *wartości specjalnego znaczenia*. W wyniku pewnych działań na wartościach rzeczywistych lub zespolonych (ale już nie całkowitych; por. także podrozdz. 14.1) może się zdarzyć, że uzyskamy czasem także *nie-liczby* (stała `NaN`, ang. ***n**ot **a** **n**umber*, wartość nieokreślona) bądź wartości *nieskończone* (stała `Inf`, ang. ***inf**inity*). Ich znaczenie jest jednak inne niż braków danych, co dość łatwo będzie nam wywnioskować z poniższych przykładów.

```
1/0              # nieskończoność (dzielenie)
## [1] Inf
0/0              # nie-liczba
## [1] NaN
sqrt(-1)         # pierwiastkowanie
## Warning in sqrt(-1): NaNs produced
## [1] NaN
log(0)           # logarytm
## [1] -Inf
(2+0i)/(0+0i)
## [1] Inf+NaNi
```

Testowanie występowania wartości specjalnych. Do sprawdzenia, które elementy wektora mają wartości nieskończone (`Inf` lub `-Inf`), służy funkcja **`is.infinite()`**. Podobnie działa **`is.nan()`** dla nie-liczb.

Nas jednak najczęściej będzie interesować funkcja **`is.finite()`**, która zwraca `TRUE` wtedy i tylko wtedy, gdy element wektora liczbowego (bądź ani część rzeczywista, ani urojona wartości zespolonej) nie jest równy `NA`, `NaN`, `Inf` ani `-Inf`.

```
x <- c(1, NA, Inf, -Inf, NaN)
is.finite(x)
## [1]  TRUE FALSE FALSE FALSE FALSE
is.infinite(x)
## [1] FALSE FALSE  TRUE  TRUE FALSE
is.nan(x)
## [1] FALSE FALSE FALSE FALSE  TRUE
is.na(x)
## [1] FALSE  TRUE FALSE FALSE  TRUE
```

2.5. Typ pusty (NULL)

Innym atomowym typem podstawowym w R, choć niebędącym już wektorem, jest NULL. W pewnym sensie możemy go postrzegać jako zbiór pusty, Ø.

Obiekt typu NULL jest dostępny pod nazwą... NULL.

```
typeof(NULL)                    # obiekt NULL jest typu podstawowego NULL
## [1] "NULL"
```

W pamięci istnieje zawsze tylko jedna instancja obiektu NULL (choć być może związana z różnymi nazwami).

WAŻNE

Pamiętajmy o tym, by nie mylić wartości NA z NULL. Pierwsza oznacza brak w danych i może stanowić element każdego wektora. Druga z nich należy zupełnie do innej dziedziny (inny typ podstawowy).

Do sprawdzenia, czy dana wartość jest typu NULL, służy funkcja **is.null()**.

```
x <- NULL
is.null(x)
## [1] TRUE
```

Zwróćmy uwagę, że każda funkcja w R musi zwracać *jakąś* wartość. Jeśli jednak celem działania funkcji jest wytworzenie tylko pewnych „skutków ubocznych" (np. narysowanie wykresu czy wypisanie komunikatu na konsoli), zwrócenie „niczego konkretnego" może być dobrym pomysłem.

```
x <- cat("Co zwraca ta funkcja?")
## Co zwraca ta funkcja?
x                               # nic ciekawego
## NULL
```

CIEKAWOSTKA

Pusty (0-elementowy) wektor nie jest tożsamy z NULL – ma on bowiem określoną informację o typie i można dlań ustalać wartości różnych atrybutów (zob. rozdz. 6).

```
is.null(numeric(0))
## [1] FALSE
is.null(c())                    # tutaj jednak brak informacji o typie
## [1] TRUE
typeof(c())
## [1] "NULL"
```

Obiekt NULL może jednak zachowywać się (gdy R będzie go rzutował do wektora) jak wektor pusty.

```
c(NULL, 1, 2, NULL, 3)
## [1] 1 2 3
length(NULL)
## [1] 0
as.numeric(NULL)
## numeric(0)
```

Gdy będziemy chcieli wykluczyć z naszych obliczeń nietypowe przypadki, takie jak „pusty wektor lub NULL", czasem napiszemy po prostu warunek `length(x)!=0`.

O zasadności korzystania z funkcji `is.null()` możemy przekonać się, studiując poniższe przykłady:

```
NULL == c()                     # porównywanie
## logical(0)
NULL == numeric(0)
## logical(0)
NULL == NA
## logical(0)
NULL == 5
## logical(0)
identical(NULL, c())
## [1] TRUE
identical(NULL, logical(0))
## [1] FALSE
```

Na takie przypadki będziemy musieli umieć się uodpornić, gdy będziemy tworzyć własne funkcje.

Poznawszy podstawowe typy atomowe, w kolejnym rozdziale omówimy najbardziej godne uwagi operacje na wektorach. Dzięki nim będziemy mogli zaimplementować pierwsze ciekawe algorytmy.

OPERACJE NA WEKTORACH 3

Czas już przyjrzeć się temu, co możemy zrobić z poznanymi w poprzednim rozdziale wektorami atomowymi. Zwróćmy baczną uwagę na wszystkie omawiane tutaj funkcje – dzięki nim będziemy mogli rozwiązać już za chwilę niemałą liczbę zagadnień obliczeniowych.

WAŻNE

Jeśli znamy jakiś strukturalny język programowania (np. C), odnotujmy, że tzw. pętle, czyli pewne wyrażenia służące do ręcznej zmiany przepływu sterowania w programach, omawiamy w tej książce znacznie dalej. Okazuje się bowiem, że w R często tego typu konstrukcji językowych da się uniknąć (z pozytywnym skutkiem dla przejrzystości, wydajności i innych aspektów jakości kodu) właśnie przez zastosowanie wybranych funkcji wbudowanych. Zmiana przyzwyczajeń wyniesionych z języków strukturalnych może być dla nas jednak pewnym wyzwaniem. W związku z powyższym samodzielne rozwiązanie towarzyszących temu rozdziałowi ćwiczeń jest więcej niż zalecane.

Programiści języka Python szybko zauważą, że znakomita większość omawianych w niniejszym rozdziale funkcji i operatorów ma swoje odpowiedniki w pakiecie NumPy; por. [29]. Podejście do programowania oparte na tzw. *wektoryzacji* (zamiast używania pętli) jest korzystne ze względów wydajnościowych także i w tym języku.

3.1. Podstawowe operatory

Przegląd funkcji wbudowanych zaczniemy od operatorów. Zasadniczo operatory możemy podzielić na dwie grupy:

- operatory binarne – tj. takie, które działają na dwóch operandach (argumentach);
- operatory unarne – stosowane na jednym operandzie.

W kolejnych podrozdziałach przyjrzymy się operatorom arytmetycznym, logicznym i relacyjnym. Poznamy również ich priorytety: wiedza o nich przyda się nam podczas budowania wyrażeń, w których występuje kilka operatorów. Dalej dowiemy się także, w jaki sposób przy użyciu operatora indeksowania możemy tworzyć ciągi złożone z wybranych elementów danych wektorów.

CIEKAWOSTKA

R jest językiem funkcyjnym: każde wykonywane *działanie* sprowadza się do wywołania pewnej funkcji. Nie inaczej jest z operatorami. Na przykład, wyrażenie `x + y` odpowiada ewaluacji wywołania `"+"(x, y)`, czyli funkcji o dość enigmatycznej na razie nazwie `"+"` na argumentach `x` i `y`. Operatory są więc specjalnym rodzajem funkcji wbudowanych: takich, które przez parser języka R są traktowane, dla naszej wygody, w szczególny sposób.

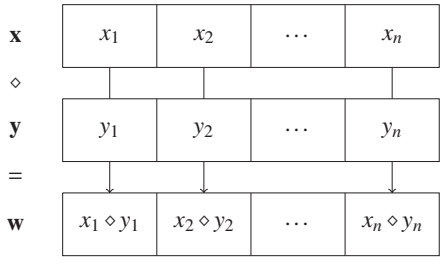
3.1.1. Operatory arytmetyczne

Do działania na wektorach liczbowych (i w niektórych przypadkach zespolonych) możemy używać binarnych operatorów arytmetycznych wymienionych w tab. 3.1.

Tabela 3.1. Operatory arytmetyczne; zob. także `?Arithmetic`

Operacja	Znaczenie
`x + y`	dodawanie
`x - y`	odejmowanie
`x * y`	mnożenie
`x / y`	dzielenie (rzeczywiste)
`x ^ y`	potęgowanie (synonim: `x ** y`)
`x %% y`	reszta z dzielenia (modulo)
`x %/% y`	dzielenie całkowite (bez reszty)

Wektoryzacja. Operatory arytmetyczne są *zwektoryzowane* (ang. *vectorized*), tzn. dla dwóch wektorów $\mathbf{x} = (x_1, \ldots, x_n)$ i $\mathbf{y} = (y_1, \ldots, y_n)$ o tej samej długości n w wyniku działania $\mathbf{x} \diamond \mathbf{y}$ otrzymujemy wektor $\mathbf{w}$ o długości n i elementach $w_i = x_i \diamond y_i$, $i = 1, \ldots, n$.



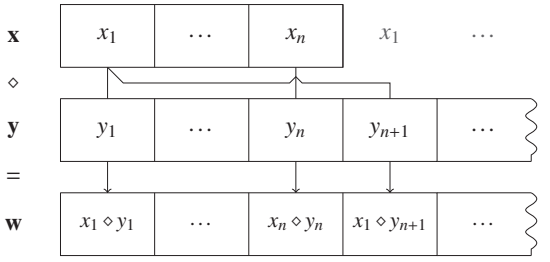
Tym samym powiemy, że operacje tego typu wykonywane są *element po elemencie* (ang. *elementwise*). Na przykład:

```
5 %% 2                    # reszta z dzielenia przez 2 (dwa wektory jednoelementowe)
## [1] 1
c(1, 2, 3) + 10:12        # równoważnie: c(1+10, 2+11, 3+12)
## [1] 11 13 15
seq(0, 2, by=0.5) %% c(1, 1, 1, 1, 1) # funkcja „podłoga"
## [1] 0.0 0.5 0.0 0.5 0.0
1:5 * 11:15 / 21:25       # łączenie operacji (zob. dalej)
## [1] 0.5238095 1.0909091 1.6956522 2.3333333 3.0000000
```

Powtórzmy, programiści wielu innych języków programowania użyliby pętli do implementacji powyższych działań. W R wcale taka potrzeba nie zachodzi. Jakaś pętla oczywiście tutaj się znajduje, ale jest ona niejako „ukryta" w kodzie operatorów (funkcji) „+", „*", czy „%%".

Reguła zawijania. Jeśli wektory będące argumentami operatorów binarnych są różnej długości, stosowana jest tzw. *reguła zawijania* (ang. *recycling rule*), która niejako „powiela" krótszy wektor, tak by uzgodnić jego długość z dłuższym wektorem.

Bardziej formalnie, niech $\mathbf{x} = (x_1, \ldots, x_n)$ i $\mathbf{y} = (y_1, \ldots, y_m)$, gdzie bez straty ogólności $1 \leqslant n \leqslant m$. Wówczas wynikiem działania $\mathbf{x} \diamond \mathbf{y}$ jest m-elementowy wektor $\mathbf{w} = (x_1 \diamond y_1, \ldots, x_n \diamond y_n, x_1 \diamond y_{n+1}, x_2 \diamond y_{n+2}, \ldots)$, tj. taki, że $w_i = x_{((i-1) \bmod n)+1} \diamond y_i$.



Powielanie następuje więc tutaj w taki sposób, jakby na krótszym z wektorów było wywoływane `rep(..., length.out=...)`.

```
2 ^ (0:5)                 # to samo, co rep(2, length.out=6)^(0:5)
## [1]  1  2  4  8 16 32
c(-1, 1) * (1:6)
## [1] -1  2 -3  4 -5  6
```

Zwróćmy uwagę, że gdy dwa wektory nie mają *zgodnych* długości, czyli gdy $m \bmod n \neq 0$, R generuje ostrzeżenie (bo może podaliśmy niezgodne argumenty przez pomyłkę):

```
c(-1, 1) * (1:5)          # ostrzeżenie (warning), to nie błąd (error)
## Warning in c(-1, 1) * (1:5): longer object length
```

```
## is not a multiple of shorter object length
## [1] -1  2 -3  4 -5
```

Ponadto, jeśli jednym z argumentów jest wektor pusty (o długości równej zeru), to zwracany jest też pusty wektor.

Uzgadnianie typów. Jeśli zestawiamy ze sobą dwa wektory różnych typów, wektor o typie mniej ogólnym jest zawsze niejawnie „promowany" do typu bardziej ogólnego przed wykonaniem operacji.

```
c(TRUE, FALSE, TRUE, FALSE) * (1:4)       # wektor logiczny → wektor całkowity
## [1] 1 0 3 0
as.integer(c(TRUE, FALSE, TRUE, FALSE)) * (1:4) # to samo
## [1] 1 0 3 0
```

Przypomnijmy, że z mechanizmem uzgadniania typów, czyli koercją, spotkaliśmy się już w poprzednim rozdziale. Warto odnotować, że wynikowy wektor jest takiego typu, jak typ najogólniejszego z podanych operandów.

```
typeof(1L * 1i)                  # wektor całkowity * zespolony → wektor zespolony
## [1] "complex"
```

Na marginesie, standardowe dzielenie wektorów o typie całkowitym zawsze daje w wyniku wektor liczb zmiennopozycyjnych.

WAŻNE

Dobrze zapamiętajmy ideę wektoryzacji, regułę zawijania i uzgadniania typów. Niebawem okaże się, że wiele innych funkcji w R również działa zgodnie z tymi zasadami. My także, pisząc w przyszłości własne funkcje, będziemy starać się je zachowywać.

Inne podobne operacje arytmetyczne. W tym miejscu warto jest wspomnieć o funkcjach `pmin()` i `pmax()`, które realizują „równoległe" minimum i maksimum dwóch lub więcej wektorów liczbowych. Choć nie mają one swojego operatorowego odpowiednika w R, operacje te czasem zapisuje się w matematyce jako, odpowiednio, $\mathbf{x} \wedge \mathbf{y}$ i $\mathbf{x} \vee \mathbf{y}$. Na przykład, dla $\mathbf{x} = (x_1, \ldots, x_n)$ i $\mathbf{y} = (y_1, \ldots, y_n)$ wywołanie `pmin`$(\mathbf{x}, \mathbf{y})$ daje w wyniku wektor $(w_1, \ldots, w_n)$, taki że $w_i = \min\{x_i, y_i\}$ dla $i = 1, \ldots, n$.

```
x <- c(5, 4, 2, 1, 3)
y <- c(3, 5, 1, 1, 6)
pmin(x, y)
## [1] 3 4 1 1 3
pmax(x, y)
## [1] 5 5 2 1 6
```

Operatory unarne. Wypada w tym miejscu wspomnieć także o unarnych operatorach arytmetycznych „+” i „−”. Prześledźmy wyniki następujących operacji:

```
+c(1, -2, 3)                    # nie robi nic ciekawego
## [1]  1 -2  3
-c(1, -2, 3)                    # zmiana znaku
## [1] -1  2 -3
```

Braki danych. Przyjrzyjmy się wynikowi operacji na wektorze, który zawiera wartość `NA`.

```
c(1, NA, 3) + 1
## [1]  2 NA  4
```

WAŻNE

Większość operacji na brakach danych daje w wyniku `NA`.

Przypomnijmy sobie także inne wartości specjalne wprowadzone w podrozdz. 2.4. Mogą pojawić się one w wyniku stosowania omawianych tutaj operatorów.

```
c(0, 1) / 0                     # dzielenie przez zero
## [1] NaN Inf
c(1 + Inf, 0 + Inf, -1000 + Inf, NaN + Inf, -Inf + Inf)
## [1] Inf Inf Inf NaN NaN
```

3.1.2. Operatory logiczne

Operatory logiczne jako argumenty przyjmują zasadniczo wektory wartości logicznych. Dają w wyniku zawsze wektor wartości logicznych. Ich wykaz zawiera tab. 3.2. Do tej grupy możemy także zaliczyć funkcję `xor()`, realizującą operację alternatywy wyłączającej (ang. *exclusive-or*).

Tabela 3.2. Operatory logiczne; zob. także `?Logic`

Operacja	Znaczenie
`!x`	negacja (unarny, ang. *not*)
`x | y`	alternatywa (ang. *or*)
`x & y`	koniunkcja (ang. *and*)

Oto wszystkie możliwe wyniki operacji logicznych, również z uwzględnieniem działań na brakach danych: