Wydział Matematyki i Nauk Informacyjnych
Politechnika Warszawska

STATYSTYKA MATEMATYCZNA

z pakietem R

I. Wprowadzenie do języka R

Przemysław Grzegorzewski
Konstancja Bobecka-Wesołowska
Marek Gągolewski

Spis treści

1 Wprowadzenie

Pakiet R [4] jest darmowym, zaawansowanym i coraz bardziej zyskującym uznanie[1] środowiskiem służącym m.in. do obliczeń statystycznych. Jego trzon stanowi wygodny, interpretowany język programowania składnią podobny do C/C++ (a nawet Perla bądź Matlaba)[2], cechujący się jednak o wiele większą siłą wyrazu. R

Program można pobrać ze strony domowej tego projektu[3]. Działa on na różnych systemach operacyjnych: Windows, Linux i MacOS, a jego instalacja na własnym komputerze jest dość łatwa i nie powinna przysporzyć kłopotów.

Do generowania przykładów w niniejszym skrypcie używamy R-a w wersji 2.9.0. Jednakże (prawie?) wszystkie ilustracje powinny działać także we wcześniejszych i późniejszych wersjach programu, jako że nie będziemy tu korzystali z zaawansowanych konstrukcji językowych ani z niestandardowych pakietów.

Osobom szczególnie zainteresowanym językiem R polecamy prace [3, 2, 5, 1].

Pierwsze, bliskie spotkanie trzego stopnia

Po uruchomieniu programu, środowisko wita użytkownika stosownym komunikatem wyświetlonym w konsoli i drukuje tzw. *znak zachęty*, standardowo:

```
>
```

oznacza to gotowość do przyjmowania poleceń od użytkownika.

By „przełamać lody", możemy najpierw potraktować R-a jako kalkulator. Wydajmy następujące polecenie, następnie zatwierdźmy je klawiszem [ENTER]. Zostanie natychmiast wyświetlony wynik.

```
> 2+2

[1] 4
```

Oczywiście doskonale pamiętamy, że w języku C istnieje wymaganie, aby każde polecenie kończyć znakiem średnika. Jak widzimy, tutaj nie jest to konieczne. Jednakże tego rodzaju separatora możemy użyć do wprowadzenia więcej niż jednego wyrażenia w jednej linii.

```
> 2*3.5; 5+4*2

[1] 7

[1] 13
```

Uwaga

Część dziesiętną ułamka oddzielamy kropką.

Inną ciekawą własnością programu jest to, że nie trzeba wpisywać całego polecenia w jednym wierszu. Jeżeli wprowadzimy niedokończone wyrażenie, to R poprosi o jego uzupełnienie w następnej linii (zwróćmy uwagę na zmianę znaku zachęty).

[1] Także w Polsce.

[2] Zakładamy u Czytelnika chociaż elementarną świadomość faktu przebycia kursów programowania na I roku studiów. Do tych głębokich (często w sensie bycia ukrytymi) pokładów wiedzy będziemy się co i raz odwoływać.

[3] http://www.r-project.org/

```
> 5 / 2 *
+ 8

[1] 20
```

Uwaga

Historia ostatnio wywoływanych poleceń dostępna jest za pośrednictwem klawiszy [↑] oraz [↓]. Możemy również dowolnie przesuwać kursor, aby np. zmienić fragment wprowadzanej komendy, podobnie jak w zwykłym edytorze tekstu. Spróbujmy użyć do tego klawiszy [←], [→], [HOME], [END].

Dobrym nawykiem jest komentowanie wpisywanego kodu. Służy do tego symbol kratki (#) — wszystkie następujące po nim znaki aż do końca linii będą ignorowane przez interpreter poleceń.

```
> 2/4  # dzielenie

[1] 0.5
```

Uwaga

Pod systemem Windows wygodnie jest używać prostego, wbudowanego edytora (menu *File* → *New script*). Można w nim robić np. notatki z zajęć lub pisać złożone funkcje. Dowolne fragmenty kodu wysyłamy do konsoli celem ich uruchomienia za pomocą kombinacji klawiszy [CTRL+R].

Pamiętajmy o częstym zapisywaniu tworzonego pliku (*File* → *Save script*)!

Jesteśmy już gotowi, aby przyjrzeć się dokładniej najważniejszym typom danych języka R.

2 Wektory

Najbardziej podstawowym typem danych, z którym będziemy mieli do czynienia, są *wektory*.

2.1 Wektory liczbowe, tworzenie wektorów, podstawowe operacje

Wywołane powyżej polecenie

```
> 2+2

[1] 4
```

jest tak na prawdę działaniem na wektorach liczbowych o długości 1, dającym w wyniku wektor o takiej samej liczbie elementów.

Do stworzenia wektora o dowolnej długości używamy funkcji `c()`.

```
> c(4,6,5,3)

[1] 4 6 5 3
```

Uwaga

Zwróćmy uwagę, że R rozróżnia wielkość liter. Znaczy, że powyższe wywołanie z użyciem nazwy C() (wielka litera) najprawdopodobniej zakończy się zgłoszeniem błędu.

Skoro pojedyncza liczba jest wektorem, to czemu by nie spróbować połączenia wektorów dłuższych w jeden?

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

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

Operacje „element po elemencie"

Większość operacji arytmetycznych na wektorach wykonywana jest *element po elemencie* (ang. *element-wise*), tzn. mając dane ciągi $\mathbf{a} = (a_1, a_2, \ldots, a_n)$ oraz $\mathbf{b} = (b_1, b_2, \ldots, b_n)$ wynikiem działania $\mathbf{a} \circledcirc \mathbf{b}$ jest wektor $(a_1 \circledcirc b_1, \ldots, a_n \circledcirc b_n)$.

```
> c(1,2,3)*c(0.5, 0.5, 0.5)

[1] 0.5 1.0 1.5
```

Reguła zawijania

Jeżeli dane operandy są różnej długości (dla ustalenia uwagi niech wynoszą one $n_1 < n_2$), następuje ich uzgodnienie za pomocą tzw. *reguły zawijania* (ang. *recycling rule*). Wektor krótszy $(a_1, a_2, \ldots, a_{n_1})$ jest powielany tylekroć tak, by dopasował się do dłuższego, wg schematu $(a_1, a_2, \ldots, a_{n_1}, a_1, a_2, \ldots)$. W przypadku, gdy rozmiary nie są zgodne ($n_2 \bmod n_1 \neq 0$), wyświetlane jest ostrzeżenie. W wyniku otrzymujemy ciąg o długości n_2. Dla przykładu, następujące wyrażenie jest tożsame z powyższym.

```
> c(1,2,3)*0.5 # to samo, co wyżej

[1] 0.5 1.0 1.5
```

Inne przykłady:

```
> c(1,2,3,4)+c(1,0.5)

[1] 2.0 2.5 4.0 4.5


> 2^c(0,1,2,3,4) # potęgowanie

[1]  1  2  4  8 16
```

Poniższa tabela zestawia dostępne operatory arytmetyczne.

Operacja	Znaczenie
`-x`	zmiana znaku
`x + y`	dodawanie
`x - y`	odejmowanie
`x * y`	mnożenie
`x / y`	dzielenie
`x ^ y`	potęgowanie
`x %/% y`	dzielenie całkowite
`x %% y`	reszta z dzielenia

Pewne operacje mogą generować wartości specjalne: nieskończoność (stała `Inf`) i nieliczbę (stała `NaN`, ang. *not a number*). `Inf`, `NaN`

```
> 1.0 / 0.0

[1] Inf


> 0.0 / 0.0

[1] NaN


> -1/Inf

[1] 0


> Inf-Inf

[1] NaN


> NaN + 100

[1] NaN


> -Inf + 100000000000

[1] -Inf
```

W język `R` wbudowanych jest również wiele funkcji matematycznych. Ich wartości wyliczane są oddzielnie dla poszczególnych elementów wektora, tzn. $f((a_1, a_2, \ldots, a_n)) = (f(a_1), f(a_2), \ldots, f(a_n))$. Funkcje matematyczne

```
> pi # to jest wbudowana stała

[1] 3.141593


> sin(c(0.0, pi*0.5, pi, pi*1.5, pi*20))

[1]  0.000000e+00  1.000000e+00  1.224647e-16 -1.000000e+00 -2.449294e-15


> round(sin(c(0.0, pi*0.5, pi, pi*1.5, pi*20)), 3); # zaokrąglijmy wynik

[1]  0  1  0 -1  0
```

Uwaga

`1.22464679914735e-16` to wynik w tzw. *notacji naukowej*. Np. liczba `3.2e-2` to nic innego niż $3{,}2 \times 10^{-2}$.

```
> 3.2e-2

[1] 0.032
```

Zatem: 10^{-16} to bardzo mała liczba (rezultat błędów arytmetyki zmiennopozycyjnej komputera), czyli prawie 0.

Wykaz najważniejszych funkcji matematycznych przedstawiamy poniżej.

Funkcja	Znaczenie
`abs(x)`	wartość bezwzględna
`sqrt(x)`	pierwiastek kwadratowy
`cos(x)`	cosinus
`sin(x)`	sinus
`tan(x)`	tangens
`acos(x)`	arcus cosinus
`asin(x)`	arcus sinus
`atan(x)`	arcus tanges
`exp(x)`	e^x
`log(x, base = exp(1))`	logarytm o podstawie `base`, domyślnie logarytm naturalny
`log10(x)`	logarytm o podstawie 10
`log2(x)`	logarytm o podstawie 2
`round(x, digits=0)`	zaokrąglanie do `digits` cyfr po kropce dziesiętnej
`floor(x)`	największa liczba całkowita nie większa niż `x`
`ceiling(x)`	najmniejsza liczba całkowita nie mniejsza niż `x`

Uwaga

W zapisie `log(x, base = exp(1))`, parametr `base` jest *parametrem domyślnym*. Jeżeli go pominiemy przy wywołaniu, zostanie zań obrana liczba, którą przewidzieli autorzy `R`-a, czyli e.

Uwaga

`R` posiada bardzo rozwinięty, wygodny i dobrze zorganizowany system pomocy. Aby uzyskać więcej informacji na temat jakiejś funkcji, np. `sin()`, piszemy

```
> ?sin
```

bądź równoważnie

```
> help(sin)
```

Możemy też skorzystać z prostej wyszukiwarki w przypadku, gdy nie znamy dokładnej nazwy pożądanej funkcji.

```
> help.search("standard deviation")
```

Ponadto, gdy piszemy polecenia w konsoli, możemy skorzystać z podpowiedzi bądź autouzupełnienia, za pomocą klawicza [TAB].

```
> ce      # tutaj wciskamy klasiwsz [TAB]...
> ceiling # ...i R "dopowiedział" nazwę funkcji
```

```
> cos    # [TAB]

cos   cosh
```

Wynika z tego, że R „zna” dwie funkcje o nazwach zaczynających się od `cos`.

Funkcje agregujące i pomocnicze

Mamy także dostęp do wielu tzw. funkcji agregujących, zwracających pojedynczą liczbę oraz dodatkowych funkcji pomocniczych, ułatwiających obliczanie różnego rodzaju wyrażeń arytmetycznych.

Funkcja	Znaczenie	Wyrażenie
`sum(x)`	suma wszystkich elementów	$r_1 := \sum_{i=1}^{n} x_i$
`prod(x)`	iloczyn wszystkich elementów	$r_1 := \prod_{i=1}^{n} x_i$
`diff(x)`	różnica sąsiadów	$r_j := x_{j+1} - x_j; \quad j = 1, 2, \ldots, n-1$
`mean(x)`	średnia arytmetyczna	$r_1 := \frac{1}{n} \sum_{i=1}^{n} x_i$
`var(x)`	wariancja	$r_1 := \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \texttt{mean(x)})^2$
`sd(x)`	odchylenie standardowe	$r_1 := \sqrt{\texttt{var(x)}} = \texttt{sqrt(var(x))}$
`sort(x)`	sortowanie ciągu	
`rank(x)`	rangowanie elementów	
`min(x)`	minimum	$r_1 := \min_{i=1,2,\ldots,n} x_i$
`max(x)`	maksimum	$r_1 := \max_{i=1,2,\ldots,n} x_i$
`cummin(x)`	minimum skumulowane	$r_j := \min_{i=1,2,\ldots,j} x_i; \quad j = 1, 2, \ldots, n$
`cummax(x)`	maksumum skumulowane	$r_j := \max_{i=1,2,\ldots,j} x_i; \quad j = 1, 2, \ldots, n$
`cumsum(x)`	skumulowana suma	$r_j := \sum_{i=1}^{j} x_i; \quad j = 1, 2, \ldots, n$
`cumprod(x)`	skumulowany iloczyn	$r_j := \prod_{i=1}^{j} x_i; \quad j = 1, 2, \ldots, n$

Zadanie 1.1. Wiedząc, że $\lim_{n\to\infty} \sqrt{\sum_{i=1}^{n} \frac{6}{i^2}} = \pi$, oblicz przybliżenie ludolfiny dla $n = 100, 1000, 10000$, oraz 100000.

Policzenie każdej z poszukiwanych wartości jest bardzo proste. Wystarczy skorzystać z operatorów arytmetycznych i funkcji `sum()` oraz `sqrt()`.

```
> sqrt(sum(6/(1:100)^2))

[1] 3.132077


> sqrt(sum(6/(1:1000)^2))

[1] 3.140638


> sqrt(sum(6/(1:10000)^2))

[1] 3.141497


> sqrt(sum(6/(1:100000)^2))

[1] 3.141583
```

□

Uwaga

Dzięki wydajnym funkcjom takim jak powyższe, w wielu wypadkach nie jest potrzebne używanie warunkowych instrukcji sterujących, takich jak np. pętla `for` z języka `C`. Celowo zatem ich wprowadzenie odwlekamy, gdyż na tym etapie nie są nam potrzebne. Mogłbyby poza tym wprowadzić złe nawyki, czego oczywiście nie chcemy.

W bardzo prosty sposób da się generować ciągi arytmetyczne. Ciąg o przyroście 1 bądź -1 tworzymy za pomocą operatora dwukropka (`:`). Ciągi arytmetyczne

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

> 1.5:6
[1] 1.5 2.5 3.5 4.5 5.5

> -1:10 # sprawdźmy tzw. priorytet operatora ":"
 [1] -1  0  1  2  3  4  5  6  7  8  9 10

> (-1):10 # tożsame z powyższym
 [1] -1  0  1  2  3  4  5  6  7  8  9 10

> -(1:10)
 [1]  -1  -2  -3  -4  -5  -6  -7  -8  -9 -10

> 5:0
[1] 5 4 3 2 1 0
```

Ciągi o innych przyrostach są konstruowane za pomocą funkcji `seq()`.

```
> seq(0, 10, 2); # od 0 do 10 co 2, podobnie:
[1]  0  2  4  6  8 10

> seq(11.5, -3, by=-3.7);
[1] 11.5  7.8  4.1  0.4

> seq(0.0, 1.0, length=5); # ciąg o ustalonej długości, przyrost wyznacza R
[1] 0.00 0.25 0.50 0.75 1.00
```

Możliwe jest także tworzenie wektorów za pomocą powtórzeń. Rozważmy następujące przykłady: Powtórzenia

```
> rep(1, 5);
[1] 1 1 1 1 1

> rep(c(1,2), 5);
 [1] 1 2 1 2 1 2 1 2 1 2

> rep(c(1,2), each=5);
 [1] 1 1 1 1 1 2 2 2 2 2

> rep(c(1,2), each=c(5,4));
 [1] 1 1 1 1 1 2 2 2 2 2

> rep(c(1,2), c(5,4));
[1] 1 1 1 1 1 2 2 2 2
```

Zadanie 1.2. Spróbuj poznać szczegóły działania tej funkcji, studiując stronę pomocy:

```
?rep
```

2.2 Wektory logiczne

Wartości logiczne

Innym typem wektorów są wektory przechowujące wartości logiczne. Mamy dostęp do dwóch wbudowanych stałych, oznaczających wartość logiczną *prawda* (`TRUE` bądź `T`) oraz wartość *fałsz* (`FALSE` bądź `F`).

Wektory tworzymy za pomocą poznanych już funkcji `c()` bądź `rep()`.

```
> c(T,F,T)

[1]  TRUE FALSE  TRUE


> rep(FALSE, 3)

[1] FALSE FALSE FALSE
```

Operacje logiczne

Mamy dostęp do następujących operacji logicznych. Działają one podobnie do operacji arytmetycznych na wektorach numerycznych, tzn. element po elemencie oraz zgodnie z regułą zawijania.

Operacja	Znaczenie
`! x`	negacja
`x & y`	koniunkcja
`x | y`	alternatywa
`xor(x, y)`	alternatywa wykluczająca

Uwaga

Zwróćmy uwagę, że operatory koniunkcji i alternatywy w języku `C` były zapisywane, odpowiednio, jako `&&` oraz `||`. W języku `R` są one także dostępne, jednakże nie działają zgodnie z zasadą element-po-elemencie, zwracając pojedynczą wartość. Odkrycie reguły ich działania pozostawiamy jako zadanie dla Czytelnika.

```
> !c(T,F)

[1] FALSE  TRUE


> c(T,F,F,T) | c(T,F,T,F)

[1]  TRUE FALSE  TRUE  TRUE


> c(T,F,F,T) & c(T,F,T,F)

[1]  TRUE FALSE FALSE FALSE


> xor(c(T,F,F,T), c(T,F,T,F))

[1] FALSE FALSE  TRUE  TRUE
```

Operatory porównania, które można stosować m.in. do ustalania związków między elementami w wektorach liczbowych, dają w wyniku ciągi wartości logicznych.

Operatory porównania

Operator	Znaczenie
`x < y`	mniejsze
`x > y`	większe
`x <= y`	mniejsze lub równe
`x >= y`	większe lub równe
`x == y`	równość
`x != y`	nierówność

```
> (1:10) <= 5

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


> (1:10) <= c(3,7) # ech, ta reguła zawijania...

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


> (1:10) %% 2 == 0 # hmm... co oznaczał ten operator arytmetyczny???

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


> rep(c(TRUE,FALSE), 4) != rep(T, 8)

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

2.3 Wektory liczb zespolonych

Mamy też możliwość tworzenia wektorów złożonych z liczb zespolonych. Jednostkę urojoną oznaczamy literą `i`. Stanowi ona przyrostek „doklejany” do liczby.

Liczby zespolone

```
> 1i

[1] 0+1i


> 3+2i

[1] 3+2i
```

Wektory zespolone można tworzyć za pomocą funkcji `c()` oraz `rep()`. Dostępne są dla nich operacje arytmetyczne `+`, `-`, `/`, `*`, `^` oraz funkcje dodatkowe (rozważmy liczbę $z = x + iy = |z|(\cos\phi + i\sin\phi)$):

Funkcja	Znaczenie	Wyrażenie
`Re(z)`	część rzeczywista	x
`Im(z)`	część urojona	y
`Mod(z)`	moduł	$\|z\| = \sqrt{x^2 + y^2}$
`Arg(z)`	argument	ϕ
`Conj(z)`	sprzężenie	$x - iy$

```
> c(1i, 2i, 3i)

[1] 0+1i 0+2i 0+3i


> rep(3+2i, 3)

[1] 3+2i 3+2i 3+2i


> (1:10)*1i # sposobem

 [1] 0+ 1i 0+ 2i 0+ 3i 0+ 4i 0+ 5i 0+ 6i 0+ 7i 0+ 8i 0+ 9i 0+10i


> (2i-2)*(3+1i)

[1] -8+4i


> (2i-2)/(3+1i)

[1] -0.4+0.8i


> Re(4i)

[1] 0


> Mod(1+1i)

[1] 1.414214
```

2.4 Wektory napisów, hierarchia typów

Napisy

Jeszcze innym ważnym typem elementów, które mogą być przechowywane w wektorach, są napisy (ang. *strings*). Definiujemy je z użyciem cudzysłowów ("...") bądź (równoważnie) apostrofów ('...'), np.

```
> 'ala ma kota'

[1] "ala ma kota"


> c("a", "kot", "ma alę")

[1] "a"      "kot"    "ma alę"


> rep("a", 4);

[1] "a" "a" "a" "a"
```

Do tej pory tworzyliśmy wektory składające się z liczb rzeczywistych, wartości logicznych, liczb zespolonych *albo* napisów. A co gdyby tak przechowywać wartości różnych typów w jednym obiekcie? Sprawdźmy.

```
> c(TRUE, 1, 1+1i, "jeden");

[1] "TRUE"  "1"     "1+1i"  "jeden"


> c(TRUE, 1, 1+1i);

[1] 1+0i 1+0i 1+1i


> c(TRUE, 1);

[1] 1 1


> c("jeden", 1, 1+1i, TRUE);

[1] "jeden" "1"     "1+1i"  "TRUE"
```

Hierarchia typów

Z przedstawionych wyżej przykładów łatwo wywnioskować, że typy danych tworzą pewną ściśle określoną hierarchię. Wszystkie elementy wektora muszą być tego samego typu. W przypadku próby stworzenia obiektu składającego się ze składowych różnych typów, ustalany jest typ wystarczający do reprezentacji wszystkich elementów, według następującej kolejności:

1. Typ logiczny.
2. Typ liczbowy.
3. Typ zespolony.
4. Napis.

Uwaga

Z formalnego punktu widzenia typ liczbowy dzielimy jeszcze na całkowity (ang. *integer*) i zmiennopozycyjny (ang. *floating point*). Liczby wprowadzane z klawiatury są traktowane jako typu zmiennopozycyjnego, nawet jeśli nie podamy jawnie ich części ułamkowej.

Warto także zauważyć, że wartość `TRUE` jest zawsze konwertowana do liczby 1, zaś `FALSE` do 0. Korzystając z tego faktu możemy szybko rozwiązać następujące zadanie.

Zadanie 1.3. Mając dany wektor logiczy sprawdzić, ile znajduje się w nim wartości `TRUE`.

Oczywiście, wystarczy w takim razie przekształcić wektor logiczny na liczbowy i wyznaczyć sumę wszystkich elementów. Okazuje się, że funkcja `sum()` sama dokonuje stosownej konwersji.

```
> sum(c(T,F,T,F,F,F,T,T,F,F));
```

```
[1] 4
```

⊡

2.5 Zmienne, indeksowanie wektorów, braki danych

Zmienne

Język programowania byłby bardzo ograniczony, gdyby nie było w nim mechanizmów zapisywania wartości do pamięci. Tego typu funkcję spełniają tutaj *zmienne*. Identyfikowane są one przez nazwę, rozpoczynającą się od litery. Nazwy mogą zawierać dowolne kombinacje liter, cyfr i znaku podkreślnika (`_`) (pamiętajmy, że `R` rozróżnia wielkość liter).

Operatory przypisania

Do przypisywania wartości do zmiennych służy jeden z trzech operatorów: `<-`, `=`, `->`. Dwa pierwsze[4] oczekują po prawej stronie wyrażenia, a po lewej nazwy zmiennej, ostatni zaś — odwrotnie.

```
> sumka <- 2+2; Sumka <- 3+3
> abs(sumka - Sumka - 1) -> roz_niczka
> roz_niczka
```

```
[1] 3
```

Widzimy, że nie trzeba deklarować zmiennej przed użyciem, jak to jest w języku `C`. Operacje przypisania domyślnie nie powodują wyświetlenia wyniku. Można jednak je wymusić za pomocą objęcia całego wyrażenia nawiasami.

```
> (zmienna <- 1:50)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
[26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

Z pewnością zaintrygowały nas wyświetlane za każdym razem przy okazji wypisywania wyniku oznaczenia „`[1]`”. W powyższym przykładzie znalazło się dodatkowo inne. Określają one indeks elementu wektora, który rozpoczyna każdy wiersz.

Indeksowanie wektorów

Wektory w języku `R`, w przeciwieństwie do `C`, są indeksowane od numeru 1. Aby poznać długość danego ciągu (a tym samym indeks ostatniego elementu), wywołujemy funkcję `length()`.

[4] Preferowany jest operator `<-`.

```
> length(-3:3);

[1] 7
```

Interesujące nas elementy znajdujące się na określonych pozycjach możemy pobrać za pomocą operatora indeksowania `[·]`.

```
> w <- -10:10;
> w[3] # trzeci element

[1] -8


> w[c(3,5)] # trzeci i piąty (ale wygoda!)

[1] -8 -6


> idx <- 4:10; w[idx]

[1] -7 -6 -5 -4 -3 -2 -1
```

Można także wykluczać pewne elementy z ciągu za pomocą odwoływania się do ujemnych indeksów:

```
> abc <- 1:10;
> abc <- abc[-4];
> abc

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


> abc[-c(3,6)]

[1]  1  2  5  6  8  9 10
```

Wektory można również indeksować za pomocą wektorów logicznych. Operator `[·]` przyjmuje wtedy jako argument ciąg o takiej samej długości co ciąg, do którego wyrazów będziemy się odwoływać (jeżeli jest on krótszy, to działa reguła zawijania). Określają one, które z elementów mają być umieszczone w podciągu wynikowym (`TRUE`), a które opuszczone (`FALSE`).

```
> x <- 1:10;
> x[c(T,F,T,F,T,T,F,F,F,F)];

[1] 1 3 5 6


> x[c(T,F)] # reguła zawijania

[1] 1 3 5 7 9


> x > 5 # wynik jest wektorem logicznym!

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

```
> x[x > 5] # cóż za siła wyrazu!
[1]  6  7  8  9 10

> (BardzoLubieRa <- x %% 3)
 [1] 1 2 0 1 2 0 1 2 0 1

> x[BardzoLubieRa == 0] # równoważnie: x[x %% 3 == 0]
[1] 3 6 9
```

Braki danych (NA)

Istnieje także specjalna stała logiczna (obok TRUE i FALSE), służąca do reprezentowania braków danych (ang. *not available data*) — NA, czyli informacji niedostępnych bądź nieznanych. Wagę jej znaczenia poznamy w rozdziale dotyczącym statystyki opisowej. Tutaj wspomnimy tylko o funkcjach pozwalających sprawdzać, czy występują braki danych w wektorach i/lub ewentualnie je usuwać.

```
> niekompletny1 <- c(T,NA,F,NA,T,T)
> c(3,4,5,NA,2,3,1) # NA ma swój odpowiednik w każdym typie
[1]  3  4  5 NA  2  3  1

> is.na(niekompletny1)
[1] FALSE  TRUE FALSE  TRUE FALSE FALSE

> niekompletny1[!is.na(niekompletny1)] # podobnie:
[1]  TRUE FALSE  TRUE  TRUE

> na.omit(niekompletny1)
[1]  TRUE FALSE  TRUE  TRUE
attr(,"na.action")
[1] 2 4
attr(,"class")
[1] "omit"

> niekompletny1[is.na(niekompletny1)] <- FALSE # zastępujemy
> niekompletny1
[1]  TRUE FALSE FALSE FALSE  TRUE  TRUE
```

Czasem mogą się nam przydać także funkcje kontrolne do sprawdzania, czy winik jest nie-liczbą (NaN), bądź czy jest skończony. Zwracają one wartości logiczne.

```
> is.nan(Inf/Inf);
[1] TRUE

> is.finite(-Inf);
[1] FALSE

> is.infinite(c(Inf/Inf, Inf*Inf, 1/Inf, Inf/1));
[1] FALSE  TRUE FALSE  TRUE
```

2.6 Wektory czynnikowe

Czynniki

Odmiennym, wyróżnionym typem wektora są obiekty do przechowywania zmiennych typu jakościowego (kategorialnego). Wartości jego elementów mogą występować tylko na z góry określonej, skończonej liczbie poziomów (z reguły kilku-kilkunastu). Są to tak zwane *wektory czynnikowe* (ang. *factors*). Każda kategoria (klasa, poziom) czynnika może być indentyfikowana za pomocą dowolnego ciągu znaków.

```
> wek <- c("Ala", "Ola", "Jola", "Ala", "Ala", "Jola", "Ala") # 3 różne wartości
> fek <- factor(wek) # konwersja na wektor czynnikowy
> fek

[1] Ala  Ola  Jola Ala  Ala  Jola Ala
Levels: Ala Jola Ola


> levels(fek) # wektor nazw poziomów

[1] "Ala"  "Jola" "Ola"


> levels(fek)[1] <- "Michał"; # można zmienić
> fek

[1] Michał Ola    Jola   Michał Michał Jola   Michał
Levels: Michał Jola Ola


> (fek2 <- factor(rep(1:5, 2)))

 [1] 1 2 3 4 5 1 2 3 4 5
Levels: 1 2 3 4 5


> levels(fek2) <- c("A", "B", "C", "D", "E"); fek2

 [1] A B C D E A B C D E
Levels: A B C D E
```

3 Listy

Listy

Zauważyliśmy wcześniej, iż wektory mogą przechowywać elementy tylko jednego typu na raz. Tego ograniczenia pozbawione są *listy*.

```
> L <- list(1, "napis", TRUE, 5:10);
> L

[[1]]
[1] 1

[[2]]
[1] "napis"

[[3]]
[1] TRUE

[[4]]
[1]  5  6  7  8  9 10
```

Dostęp do poszczególnych elementów listy uzyskujemy za pomocą podwójnych nawiasów kwadratowych:

```
> L[[1]]

[1] 1


> L[[4]]

[1]  5  6  7  8  9 10


> L[[4]][3] # trzeci element wektora będącego 4 elementem listy

[1] 7
```

Poszczególne elementy listy mogą być także nazwane (identyfikowane za pomocą nazwy):

Elementy nazwane

```
> el <- list(1, komunikat="napis", TRUE, wartosci=5:10); el

[[1]]
[1] 1

$komunikat
[1] "napis"

[[3]]
[1] TRUE

$wartosci
[1]  5  6  7  8  9 10
```

Dostęp do drugiego elementu możemy teraz uzyskać na jeden z trzech sposobów:

```
> el[[2]]

[1] "napis"


> el$komunikat

[1] "napis"


> el["komunikat"]

$komunikat
[1] "napis"
```

Nazwy elementów można zmieniać za pomocą atrybutu `names()`. Zwraca on wektor nazw wszystkich elementów.

```
> names(el)

[1] ""          "komunikat" ""          "wartosci"
```

```
> names(el)[1:2] <- c("jedynka", "uwaga");
> el

$jedynka
[1] 1

$uwaga
[1] "napis"

[[3]]
[1] TRUE

$wartosci
[1]  5  6  7  8  9 10
```

Uwaga

W podobnym sposób można też nazywać elementy wektorów.

4 Ramki danych

Ramki danych

Szczególnego rodzaju listami są *ramki danych* (ang. *data frames*). Są to listy przechowujące wektory o tej samej długości. Przechowywane dane wyświetlane są w postaci dwuwymiarowej tabeli, w której kolumnami (zmiennymi) są wspomniane wektory, a wierszami (obserwacjami) — elementy wektorów o tych samych indeksach.

```
> imiona <- c("Ania", "Kasia", "Janek", "Borys")
> wiek <- c(8, 5, 3, 9)
> lubiaLody <- c(T, T, F, T)
> dzieci <- data.frame(imiona, wiek, lubiaLody)
> dzieci

  imiona wiek lubiaLody
1   Ania    8      TRUE
2  Kasia    5      TRUE
3  Janek    3     FALSE
4  Borys    9      TRUE
```

Domyślnie nazwy kolumn biorą się z nazw argumentów. Możemy je jednak zainicjować w podobny sposób jak w przypadku listy bądź zmienić za pomocą `names()`.

```
> dzieci <- data.frame(imie=imiona, wiek, lubiaLody); dzieci

   imie wiek lubiaLody
1  Ania    8      TRUE
2 Kasia    5      TRUE
3 Janek    3     FALSE
4 Borys    9      TRUE


> names(dzieci)[3] <- "lubiLody"; dzieci
```

```
   imie wiek lubiLody
1  Ania    8     TRUE
2 Kasia    5     TRUE
3 Janek    3    FALSE
4 Borys    9     TRUE
```

Spójrzmy, w jaki sposób uzyskać dostęp do poszczególnych fragmentów danych:

```
> dzieci[1,1] # pojedyncza komórka

[1] Ania
Levels: Ania Borys Janek Kasia


> dzieci[2:4, c(1,3)]

   imie lubiLody
2 Kasia     TRUE
3 Janek    FALSE
4 Borys     TRUE


> dzieci[1,] # pierwszy wiersz  (brak wartości w zakresie = cały zakres)

  imie wiek lubiLody
1 Ania    8     TRUE


> dzieci[,1] # pierwsza kolumna (brak wartości w zakresie = cały zakres)

[1] Ania  Kasia Janek Borys
Levels: Ania Borys Janek Kasia


> dzieci[1] # pierwsza kolumna - nazwana

   imie
1  Ania
2 Kasia
3 Janek
4 Borys


> dzieci$imie

[1] Ania  Kasia Janek Borys
Levels: Ania Borys Janek Kasia
```

Uwaga

Widzimy, że kolumna `imie` jest typu czynnikowego. Łancuchy znaków są przy tworzeniu ramek danych domyślnie automatycznie konwertowane — dla wygody — na czynniki. Możemy temu zapobiec poprzez ustawienie odpowiedniego parametru funkcji `data.frame()`:

```
> dzieci2 <- data.frame(imiona, wiek, lubiaLody, stringsAsFactors=F)
> dzieci2[,1] # napisy

[1] "Ania"  "Kasia" "Janek" "Borys"
```

5 Macierze

Macierz

Ostatnim (w kolejności, lecz nie pod względem znaczenia praktycznego) typem danych, który omówimy, są *macierze* (ang. *matrices*). Do ich tworzenia służy funkcja `matrix()`. Jako parametr pobiera ona wektor pożądanych elementów oraz rozmiar.

```
> matrix(1:6, nrow=2, ncol=3) # dwa wiersze, trzy kolumny

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


> matrix(1:6, nrow=2) # dwa wiersze także, liczbę kolumn sam sobie wyznaczy R

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


> matrix(1:6, nrow=2, byrow=T) # wprowadzamy dane wierszami

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

Operacje i funkcje macierzowe

Listę ważniejszych operacji i funkcji macierzowych przedstawia poniższa tabela.

Operacja	Znaczenie
`A %*% B`	mnożenie macierzy
`det(A)`	wyznacznik macierzy
`t(A)`	transpozycja
`solve(A, b)`	rozwiązanie układu liniowego postaci $\mathbf{Ax} = \mathbf{b}$
`diag(A)`	diagonala (jako wektor)
`eigen(A)`	wartości własne i wektory własne

```
> (A <- matrix(c(2,0,0,2), nrow=2));

     [,1] [,2]
[1,]    2    0
[2,]    0    2


> (B <- matrix(1:4, nrow=2));

     [,1] [,2]
[1,]    1    3
[2,]    2    4


> A+B   # element po elemencie
```

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

```
> A*B   # element po elemencie
```

```
     [,1] [,2]
[1,]    2    0
[2,]    0    8
```

```
> A%*%B # mnożenie macierzy
```

```
     [,1] [,2]
[1,]    2    6
[2,]    4    8
```

```
> det(B)
```

```
[1] -2
```

```
> eigen(B) # to akurat łatwo policzyć w pamięci
```

```
$values
[1]  5.3722813 -0.3722813

$vectors
           [,1]       [,2]
[1,] -0.5657675 -0.9093767
[2,] -0.8245648  0.4159736
```

```
> t(B)
```

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

6 Zadania do rozwiązania

Zadanie 1.4. Na czym polega reguła zawijania dla wektorów?

Zadanie 1.5. Utwórz tabelkę działań dla podstawowych operatorów logicznych dla wszystkich kombinacji wartości `TRUE`, `FALSE` oraz `NA`.

Zadanie 1.6. Wspomnieliśmy, iż w jezyku `R` dostępne są operatory `&&` oraz `||`. Na jakiej zasadzie działają? Spróbuj ją odkryć testując operacje na różnych wektorach logicznych.

⋆ **Zadanie 1.7.** Indeksem Hirscha dla uporządkowanego nierosnąco ciągu $\mathbf{C} = (c_1, c_2, \ldots, c_n)$, $c_i \geqslant c_j$ dla $i \leqslant j$, $c_1 > 0$, nazywamy wartość

$$h(\mathbf{C}) = \max\{i : c_i \geqslant i\} = \sum_{i=1}^{n} \mathbf{1}(c_i \geqslant i), \tag{1}$$

gdzie $\mathbf{1}(w)$ oznacza tzw. funkcję indykatorową, przyjmującą wartość 1, jeżeli warunek w jest spełniony, oraz 0 — w przeciwnym przypadku. Wyznacz za pomocą `R`-a wartość indeksu Hirscha np. dla następujących wektorów: $(43, 12, 9, 4, 3, 2, 0, 0)$, $(1, 1, 1, 1, 1, 1)$, $(32, 74, 24, 64, 123, 6, 0, 35, 1, 1, 1, 3, 64, 0, 0)$.

⋆ **Zadanie 1.8.** Indeksem Egghego dla uporządkowanego nierosnąco ciągu $\mathbf{C} = (c_1, c_2, \ldots, c_n)$, $c_i \geqslant c_j$ dla $i \leqslant j$, $c_1 > 0$, nazywamy wartość

$$g(\mathbf{C}) = \max\left\{i : \sum_{j=1}^{i} c_j \geqslant i^2\right\} = \sum_{i=1}^{n} \mathbf{1}\left(\sum_{j=1}^{i} c_j \geqslant i^2\right). \tag{2}$$

Wyznacz za pomocą `R`-a wartość indeksu g dla różnych wektorów.

7 Wskazówki

Wskazówka do zadania 1.7. W przypadku danych ciągów nieuporządkowanych, posortuj je funkcją `sort()` z odpowiednim parametrem. Skorzystaj z funkcji `sum()`.

Wskazówka do zadania 1.8. Użyj m.in. funkcji `cumsum()`.

Bibliografia

[1] Przemysław Biecek. *Przewodnik po pakiecie `R`.* OW GiS, Wrocław, 2008.

[2] Michael J. Crawley. *The `R` Book.* Wiley, 2008.

[3] `R` Development Core Team. *An Introduction to `R`*, 2009. http://cran.r-project.org/doc/manuals/R-intro.html.

[4] `R` Development Core Team. *`R`: A Language and Environment for Statistical Computing.* `R` Foundation for Statistical Computing, Vienna, Austria, 2009. http://www.R-project.org.

[5] Łukasz Komsta. *Wprowadzenie do środowiska `R`*, 2004. http://cran.r-project.org/doc/contrib/Komsta-Wprowadzenie.pdf.