Wprowadzenie do R

Dylewicz, Kamchen, Krasoń, Kulon, Soból

12 01 2021

Podstawowe typy danych

Język R posiada kilka typów danych, które pokrótce postaramy sie omówić poniżej. Pokażemy ich budowe jak i operacje na nich, przytaczając stosowne przyklady.

Liczby

Liczby całkowite i rzeczywiste (tutaj separator dziesiętny to kropka). Możemy używać również notacji naukowej. Operacje na liczbach to podstawowe działania matematyczne jak i trochę rozszerzone, ukazane niżej wraz z specjalnymi liczbami.

```
5; 5.5; 5.5e-2;
```

```
## [1] 5
```

```
## [1] 5.5
```

```
## [1] 0.055
```

Tutaj liczby specjalne,

```
NaN # not a number
```

```
## [1] NaN
```

```
Inf # nieskończoność
```

```
## [1] Inf
```

```
-Inf # - nieskończoność
```

```
## [1] -Inf
```

oraz kilka działań na liczbach

```r
1 + 1 # podobnie '-' to odejmowanie
```

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

```r
4/2 # dzielenie, a '*' to mnożenie
```

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

```r
5 %/% 3 # dzielenie całkowite
```

```
## [1] 1
```

```r
5 %% 3 # reszta z dzielenia
```

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

```r
2^3 # potęgowanie
```

```
## [1] 8
```

```r
2**3 # też potęgowanie
```

```
## [1] 8
```

```r
sqrt(4) #pierwiastkowanie
```

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

```r
abs(-1) # wartość bezwzględna
```

```
## [1] 1
```

Łańcuchy znaków

Łańcuch znaków to po prostu napi. Napis jest otoczony przez " lub '. W napisie możemy umieszczać dowolne znaki, pamiętając że są też znaki specjalne (rozpoczynające się od \ i mające specjalne funkcje). Na napisach istnieje wiele operacji (np. `paste()`, czyli sklejenie dwóch napisów), lecz je zobaczymy w notatce o napisach.

```r
"napis"
```

```
## [1] "napis"
```

```r
'to też'
```

```
## [1] "to też"
```

```
"'a tutaj nawet z bonusem'"
```

```
## [1] "'a tutaj nawet z bonusem'"
```

```
# ""a"" to już wbrew intuicji nie jest napis
cat("i znak \n specjalny, wstawiający nową linie") # cat() wyświetla napis w sposób niesformatowany
```

```
## i znak
##  specjalny, wstawiający nową linie
```

Wartości logiczne

Logiczna Prawda (`TRUE` lub `T`) oraz logiczny Fałsz (`FALSE` lub `F`). Na tych obiektach możemy wykonywać operacje logiczne oraz algebraiczne.

```
TRUE & TRUE # operator 'i'
```

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

```
TRUE | FALSE # operator 'lub'
```

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

```
1 == 1 # testowanie równości
```

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

```
1 != 2 # testowanie nierówności
```

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

```
2*TRUE # TRUE ma wartość 1
```

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

```
2*FALSE # FALSE ma wartość 0
```

```
## [1] 0
```

```
T ; `T` <- FALSE; T # używając `` możemy zmienić wartość logiczną wyrażenia
```

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

```
## [1] FALSE
```

Wektory

Wektor to w R uporządkowany zbiór elementów. Elementy te muszą mieć ten sam typ, także jeśli do wektora trafią elementy z różnym typem (poza NA), to nastąpi konwersja elementów do jednego typu. Proste wektory tworzymy przez polecenie `c()` i elementy wypisujemy w nawiasie po przecinku. Dodatkowo, element wektora jest traktowany jako jednoelementowy wektor. Wektory liczbowe jak i inne możemy tworzyć za pomocą wbudowanych funkcji do tego przeznaczonych.

```
v <- c(1, 2, 3) #przypisanie wektora do zmiennej
0:10 # wektor liczbowy
```

```
##  [1]  0  1  2  3  4  5  6  7  8  9 10
```

```
seq(from = 0, to = 10, by = 1) # to samo, ale za pomocą seq(), czyli sequance
```

```
##  [1]  0  1  2  3  4  5  6  7  8  9 10
```

```
seq(0, 1, length.out = 4) # równe odstępy w 4 liczbowym wektorze
```

```
## [1] 0.0000000 0.3333333 0.6666667 1.0000000
```

```
length(v) # zwraca długość vectora
```

```
## [1] 3
```

```
# vector(mode, lenght) tworzy wektor dlugosci lenght, a wyrazy tego wektora maja klase mode
vector("integer", 10) # wektor liczb calkowitych
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0
```

```
vector("numeric", 10) # wektor liczb rzeczywistych
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0
```

```
vector("character", 10) # wektor słów
```

```
##  [1] "" "" "" "" "" "" "" "" "" ""
```

```
rep(v, each = 2) # każdy element v zostanie powtórzony 2 razy
```

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

```
rep(v, times = 2) # v zostanie powtórzony 2 razy
```

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

```r
# mały mix tj. tutaj element v traktujemy jako wektor jednoelementowy
# i powtarzamy times razy
rep(v, times = 1:3)
```

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

```r
x <- c("a", "A") # wektor napisowy
v <- "a" # to też
toupper(x) # zmieni stringi w argumencie na wielkie litery
```

```
## [1] "A" "A"
```

```r
tolower(x) # zmieni stringi w argumencie na male litery
```

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

Indeksowanie

W R wektory są indeksowane od 1 (a nie od 0 jak w wielu językach programowania!). Aby odwołać się do konkretnego elementu wektora korzystamy z nawiasów kwadratowych `[]`.

```r
letters[3]
```

```
## [1] "c"
```

Można wybrać więcej niż jeden element, wpisując w nawiasach kwadratowych wektor indeksów.

```r
letters[1:10]
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

```r
letters[c(1, 15)]
```

```
## [1] "a" "o"
```

```r
letters[seq(1, 20, by = 2)]
```

```
##  [1] "a" "c" "e" "g" "i" "k" "m" "o" "q" "s"
```

Jeśli przed wektorem indeksów widnieje znak minus, R zwróci wszystkie elementy wektora z wyjątkiem tych w nawiasie kwadratowym.

```r
letters[-(1:10)] # niezbędny nawias wokół 1:10
```

```
##  [1] "k" "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```
letters[-c(1, 15)]
```

```
##  [1] "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "p" "q" "r" "s" "t" "u"
## [20] "v" "w" "x" "y" "z"
```

```
letters[-seq(1, 20, by = 2)]
```

```
##  [1] "b" "d" "f" "h" "j" "l" "n" "p" "r" "t" "u" "v" "w" "x" "y" "z"
```

Pod wybrane indeksy można przypisać nowe wartości.

```
new_letters <- letters
new_letters[1:5] <- LETTERS[1:5]
new_letters
```

```
##  [1] "A" "B" "C" "D" "E" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
```

Albo pod każdy wybrany indeks nową wspólną wartość.

```
new_letters[1:5] <- "x"
new_letters
```

```
##  [1] "x" "x" "x" "x" "x" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
```

Tworząc wektor funkcją `c()`, możemy nazwać każdy z jego elementów.

```
str_vec_nam <- c("a" = "A", "b" = "B", "c" = "C")
str_vec_nam
```

```
##   a   b   c
## "A" "B" "C"
```

Może być to użyteczne przy odwoływaniu się do konkretnego elementu wektora, nie trzeba wtedy znać numeru jego indeksu.

```
str_vec_nam["a"]
```

```
##   a
## "A"
```

```
str_vec_nam[c("a", "c")]
```

```
##   a   c
## "A" "C"
```

```
str_vec_nam[c("c", "a")]
```

```
##   c   a
## "C" "A"
```

Wektory możemy również indeksować za pomocą wektorów logicznych. Działa to wtedy jak wybieranie tych elementów wektora, które spełniają ustalony warunek.

```
x_ind <- new_letters == "x"
x_ind
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
## [25] FALSE FALSE
```

```
new_letters[x_ind]
```

```
## [1] "x" "x" "x" "x" "x" "x"
```

```
seq_vec <- seq(0, 1, length.out = 10)
seq_vec[seq_vec < 0.5]
```

```
## [1] 0.0000000 0.1111111 0.2222222 0.3333333 0.4444444
```

Można oczywiście rozbudowywać wyrażenia logiczne, np. następująco:

```
seq_vec[seq_vec < 0.3 | seq_vec > 0.8]
```

```
## [1] 0.0000000 0.1111111 0.2222222 0.8888889 1.0000000
```

```
seq_vec[seq_vec > 0.3 & seq_vec < 0.8]
```

```
## [1] 0.3333333 0.4444444 0.5555556 0.6666667 0.7777778
```

Operacje na wektorach

W R domyślnym i naturalnym zachowaniem funkcji na wektorach jest działanie element po elemencie

```
1:10 + seq(0, 1, length.out = 10)
```

```
##  [1]  1.000000  2.111111  3.222222  4.333333  5.444444  6.555556  7.666667
##  [8]  8.777778  9.888889 11.000000
```

```
c(2,4,6,8)^(1:4)
```

```
## [1]    2   16  216 4096
```

W przypadku gdy wektory, na których wykonujemy obliczenia mają różne długości zachodzi recykling, tj. R samoistnie przedłuża krótszy wektor replikując go odpowiednią liczbę razy. Widzimy, że obie poniższe linie kodu dają taki sam efekt.

```r
1:10 + 1:2
```

```
##  [1]  2  4  4  6  6  8  8 10 10 12
```

```r
1:10 + rep(1:2, 5)
```

```
##  [1]  2  4  4  6  6  8  8 10 10 12
```

Gdy długość dłuższego wektora nie jest wielokrotnością krótszego, recykling także zadziała, jednak R zgłosi warning.

```r
1:10 + 1:3
```

```
## Warning in 1:10 + 1:3: długość dłuszego obiektu nie jest wielokrotnością
## długości krótszego obiektu
```

```
##  [1]  2  4  6  5  7  9  8 10 12 11
```

```r
1:10 + 1:3 + 1:2 + 1:5
```

```
## Warning in 1:10 + 1:3: długość dłuszego obiektu nie jest wielokrotnością
## długości krótszego obiektu
```

```
##  [1]  4  8 10 11 13 12 11 15 17 18
```

Na wektorach możemy wykonywać oczywiście inne funkcje poza podstawowymi operacjami arytmetycznymi. Jedną z opcji jest posortowanie wektora.

```r
num_vec <- c(3,6,1,9,8,-3,0,102,-5)
sort(num_vec) # sortowanie rosnące
```

```
## [1]  -5  -3   0   1   3   6   8   9 102
```

```r
sort(num_vec, decreasing = TRUE) # sortowanie malejące
```

```
## [1] 102   9   8   6   3   1   0  -3  -5
```

Odwrócić kolejnośc elementów wektora można następująco.

```r
rev(num_vec)
```

```
## [1]  -5 102   0  -3   8   9   1   6   3
```

Oto kilka kolejnych funkcji.

```r
sum(num_vec) # suma elementów wektora
```

```
## [1] 121
```

```r
prod(num_vec) # iloczyn elementów wektora
```

```
## [1] 0
```

```r
mean(num_vec) # średnia elementów wektora
```

```
## [1] 13.44444
```

Przy operacjach jak powyższe należy jednak uważać na wektory zawierające "NA".

```r
vec_with_NA <- c(3,6,1,NA)
sum(vec_with_NA)
```

```
## [1] NA
```

Aby zsumować wartości z pominięciem "NA" należy dopisać dodatkowy argument funkcji.

```r
sum(vec_with_NA, na.rm = TRUE)
```

```
## [1] 10
```

Analogicznie dla iloczynu i średniej elementów.

```r
prod(vec_with_NA)
```

```
## [1] NA
```

```r
prod(vec_with_NA, na.rm = TRUE)
```

```
## [1] 18
```

```r
mean(vec_with_NA)
```

```
## [1] NA
```

```r
mean(vec_with_NA, na.rm = TRUE)
```

```
## [1] 3.333333
```

Listy

Lista jest podobna do wektora tj. jest pewnym ciągiem obiektów, tyle że jej elementy mogą mieć różne typy.

Tworzenie list

```r
l <- list(1:5)
#lista z elementami bedacymi liczbami
```

```
## [[1]]
## [1] 1 2 3 4 5
```

```r
l2 <- list(zwierze='dog', imie='Max',czyLubiInnePsy = TRUE)
#lista z elementami bedacymi stringami lub wartosciami logicznymi
```

```
## $zwierze
## [1] "dog"
##
## $imie
## [1] "Max"
##
## $czyLubiInnePsy
## [1] TRUE
```

Odwoływanie się do elementów list

Kolejnę różnica pomiedzy wektorem a listą jest możliwość odwoływania się do elementów listy za pomocą nazwy tego elementu i operatora $. Np:

```r
# odwolanie do elementu bedacego za pomoca [],
# wynikiem takiej operacji jest lista zawierajaca wektor
l[1]
```

```
## [[1]]
## [1] 1 2 3 4 5
```

```r
# aby odwolac sie do konkretnego elementu uzwywamy [[]], na przyklad operacja l[[1]][2]
# zwroci drugi element wektora z listy
 l[[1]][2]
```

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

```r
# nadpisywanie elementu listy wektorem
l[[1]] <- c("a", "b", "c")
# odwolanie do elementu za pomoca nazwy elementu
l2$zwierze
```

```
## [1] "dog"
```

```r
l2$imie
```

```
## [1] "Max"
```

```
l2$czyLubiInnePsy
```

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

Operacje na listach

Listy można łączyć oraz modyfikować. Funkcja `lapply()` to funkcja, która pozwala na wykonanie pewnego konkretnego działania na KAŻDYM elemencie z listy. Na przykład, możemy każdy element chcieć zapisać tylko dużymi literami:

```
lapply(l2,toupper)
```

```
## $zwierze
## [1] "DOG"
##
## $imie
## [1] "MAX"
##
## $czyLubiInnePsy
## [1] "TRUE"
```

Aby połączyć dwie listy, należy użyć `c()`, robiąc z dwóch list wektor i przypisując go do nowej zmiennej.

```
l3 <- c(l,l2)
```

```
## [[1]]
## [1] "a" "b" "c"
##
## $zwierze
## [1] "dog"
##
## $imie
## [1] "Max"
##
## $czyLubiInnePsy
## [1] TRUE
```

Macierze

Macierz to obiekt dwuwymiarowy. Składa się z elementów tego samego typu. Tworzy się ją funkcją `matrix()`, do której podajemy wartości macierzy (zwykle w postaci wektora), liczbę wierszy i kolumn.

```
matrix(data = 1:10, nrow = 2, ncol = 5)
```

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

Widzimy, że R domyślnie wypełnia macierz po kolumnach. Aby wypełnić ją po wierszach ustalamy parametr `byrow = TRUE`

```
m <- matrix(data = 1:10, nrow = 2, ncol = 5, byrow = TRUE)
m
```

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

Elementy macierzy wybiera się za pomocą dwóch indeksów - indeksu wiersza i indeksu kolumny umieszczonych w nawiasach kwadaratowych i rozdzielonych przecinkiem.

```
m[2,3]
```

```
## [1] 8
```

Można również wybrać konkretne wiersze lub kolumny.

```
m[1:2,3:4] # wybiera wiersze 1 i 2 oraz kolumny 3 i 4
```

```
##      [,1] [,2]
## [1,]    3    4
## [2,]    8    9
```

```
m[2,c(1,4,5)] # wybiera wiersz 2 oraz kolumny 1,4 i 5
```

```
## [1]  6  9 10
```

Nie podanie indeksu przed przecinkiem oznacza, że chcemy otrzymać wszystkie wiersze. Analogicznie nie podanie indeksu po przecinku oznacza, że chcemy otrzymać wszystkie kolumny.

```
m[,c(1,3)]
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    6    8
```

```
m[2,]
```

```
## [1]  6  7  8  9 10
```

Macierze, podobnie jak wektory, możemy także indeksować warunkami logicznymi.

```
# zwraca elementy (w tym wypadku element) z pierwszej kolumny,
# które są większe od 2
m[m[,1] > 2, 1]
```

```
## [1] 6
```

Można także indeksować macierz inną macierzą o dwóch kolumnach. Zwrócone zostaną wtedy elementy o indeksach będących wierszami tej macierzy.

```
matrix_ind<- matrix(c(1, 2, 2, 3, 2, 4), byrow = TRUE, nrow = 3, ncol = 2)
m[matrix_ind]
```

```
## [1] 2 8 9
```

Na macierzach o tych samych wymiarach możemy wykonywać operacje arytmetyczne. Trzeba zwrócić uwagę, że są one wykonywane element po elemencie (z matematycznego punktu widzenia jest to oczekiwane przy dodawaniu, ale nieoczekiwane przy mnożeniu macierzy).

```
m1 <- matrix(1:4,2,2)
m1
```

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

```
m2 <- matrix(2:5,2,2)
m2
```

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

```
m1 + m2
```

```
##      [,1] [,2]
## [1,]    3    7
## [2,]    5    9
```

```
m1 * m2
```

```
##      [,1] [,2]
## [1,]    2   12
## [2,]    6   20
```

Aby wykonać matematyczne mnożenie macierzy należy użyć operatora **%*%**.

```
m1 %*% m2
```

```
##      [,1] [,2]
## [1,]   11   19
## [2,]   16   28
```

Ramki danych

Jest to obiekt przechowujący dane w postaci tabeli dwuwymiarowej, którą tworzą wektory o dowolnym typie. Z ramki danych można korzystać jak z macierzy dwuwymiarowej (poprzez korzystanie z `[,]`), jak i z listy (poprzez korzystanie z $).

Tworzenie ramek danych

```r
imie <- c("Max", "Reksio","Rex","Luna") #utworzymy ramke z 2 wektorow
wiek <- c(2,8,3,11)
ramka <- data.frame(imie,wiek)
 #ramke tworzymy za pomoca polecenia data.frame()
```

```
##     imie wiek
## 1    Max    2
## 2 Reksio    8
## 3    Rex    3
## 4   Luna   11
```

```r
#wyswietlanie nazw kolumn
names(ramka)
```

```
## [1] "imie" "wiek"
```

Odwoływanie się do elementów ramek danych

```r
#odnoszenie sie do elementu znajdujacego sie w 2. rzedzie i 1. kolumnie
ramka[2,1]
```

```
## [1] "Reksio"
```

```r
#pobieranie paru wierszy na raz za pomoca wektora
ramka[c(1, 2), ]
```

```
##     imie wiek
## 1    Max    2
## 2 Reksio    8
```

```r
#pobieranie wszystkich kolumn dla 1. wiersza
ramka[1,]
```

```
##   imie wiek
## 1  Max    2
```

```r
#pobieranie wszystkich wierszy dla 1. kolumny
ramka[,1]
```

```
## [1] "Max"    "Reksio" "Rex"    "Luna"
```

```r
# pierwsza kolumna bez drugiego wiersza
ramka[-2, 1]
```

```
## [1] "Max"  "Rex"  "Luna"
```

```r
#pobieranie kolumn/wierszy po nazwie
ramka$wiek
```

```
## [1]  2  8  3 11
```

```r
# inny sposób indeksowanie po nazwie
ramka[, "wiek"]
```

```
## [1]  2  8  3 11
```

- Indeksowanie na podstawie zawartości ramki danych
  Dane z ramki mogą być przez nas "filtrowane" za pomocą []. Na przykład

```r
# psy poniżej 9 roku życia
ramka[ramka$wiek < 9, ]
```

```
##     imie wiek
## 1    Max    2
## 2 Reksio    8
## 3    Rex    3
```

```r
#dane tylko dla Reksia
ramka[ramka$imie == "Reksio", ]
```

```
##     imie wiek
## 2 Reksio    8
```

```r
# analogicznie dla wektorów
wiek[wiek < 9]
```

```
## [1] 2 8 3
```

Operacje na ramkach danych

Tworząc ramkę danych należy pamiętać o tym, aby wektory danych służące za kolumny były tej samej długości.

```r
#zamiana nazw kolumn
names(ramka) <- c("imie_psa", "wiek_psa")
```

```
##   imie_psa wiek_psa
## 1      Max        2
## 2   Reksio        8
## 3      Rex        3
## 4     Luna       11
```

Ramki danych możemy powiększać o dodatkowe wiersze i kolumny, ale typy (dla wierszy) i rozmiary muszą sie zgadzać z typami i rozmiarem ramki danych. Rozpatrzmy poniższy przykład, aby pokazać, jak dodać wiersz i kolumnę za pomocą funkcji `cbind()` oraz `rbind()`.

```r
#dodawanie nowego wiersza
dodajemy_wiersz <- data.frame(imie_psa ="Quentin", wiek_psa=9)
#funkcja rbind "skleja" wierszowo argument pierwszy (u nas ramka) z drugim
ramka <- rbind(ramka,dodajemy_wiersz)
#dodawanie nowej kolumny
czyLubiInnePsy <- c(TRUE,TRUE, FALSE, TRUE, FALSE)
#funkcja cbind "skleja" kolumnowo argument pierwszy (u nas ramka) z drugim
ramka <- cbind(ramka,czyLubiInnePsy)
```

```
##   imie_psa wiek_psa czyLubiInnePsy
## 1      Max        2           TRUE
## 2   Reksio        8           TRUE
## 3      Rex        3          FALSE
## 4     Luna       11           TRUE
## 5  Quentin        9          FALSE
```

Możemy rownież dodawać wiersze za pomocą indeksowania, to znaczy przypisywania wartości do konkretnych indeksów ramki:

```r
#jako 6. wiersz "wkladamy" nowy wektor
ramka[6,] <- c("Fanta",0.5,TRUE)
```

```
##   imie_psa wiek_psa czyLubiInnePsy
## 1      Max        2           TRUE
## 2   Reksio        8           TRUE
## 3      Rex        3          FALSE
## 4     Luna       11           TRUE
## 5  Quentin        9          FALSE
## 6    Fanta      0.5           TRUE
```

```r
# jako 4.kolumne "wkladamy" nowy wektor
ramka[,4] <- c("Mateusz","Romek","Renata","Leon","Quennie","Filip")
# nazywamy kolumne 4.
names(ramka)[4] <- "opiekun_psa"
```

```
##   imie_psa wiek_psa czyLubiInnePsy opiekun_psa
## 1      Max        2           TRUE     Mateusz
## 2   Reksio        8           TRUE       Romek
## 3      Rex        3          FALSE      Renata
## 4     Luna       11           TRUE        Leon
## 5  Quentin        9          FALSE     Quennie
## 6    Fanta      0.5           TRUE       Filip
```

Badanie ramek

Analizując nową dla nas ramkę danych, użyteczne okazują się funkcje pozwalające na poznanie właściwości ramki danych. Oto pare z nich:

```r
# wymiary ramki (6 wierszy,4 kolumny) mozna sprawdzic za pomoca funkcji dim()
dim(ramka)
```

```
## [1] 6 4
```

```
# aby zobaczyc skrocony opis typow danych zawartych w ramce uzywana jest funkcja str()
str(ramka)
```

```
## 'data.frame':    6 obs. of  4 variables:
##  $ imie_psa      : chr  "Max" "Reksio" "Rex" "Luna" ...
##  $ wiek_psa      : chr  "2" "8" "3" "11" ...
##  $ czyLubiInnePsy: chr  "TRUE" "TRUE" "FALSE" "TRUE" ...
##  $ opiekun_psa   : chr  "Mateusz" "Romek" "Renata" "Leon" ...
```

```
# aby "podejrzec" pierwsze wiersze ramki danych, wraz naglowkami kolumn uzywana jest funkcja head()
head(ramka)
```

```
##   imie_psa wiek_psa czyLubiInnePsy opiekun_psa
## 1      Max        2           TRUE     Mateusz
## 2   Reksio        8           TRUE       Romek
## 3      Rex        3          FALSE      Renata
## 4     Luna       11           TRUE        Leon
## 5  Quentin        9          FALSE     Quennie
## 6    Fanta      0.5           TRUE       Filip
```

```
# wysietlanie pierwszych n wierszy
head(ramka,n=2)
```

```
##   imie_psa wiek_psa czyLubiInnePsy opiekun_psa
## 1      Max        2           TRUE     Mateusz
## 2   Reksio        8           TRUE       Romek
```

```
# wyswietlanie ostatnich n wierszy za pomoca funkcji tail()
tail(ramka,n=2)
```

```
##   imie_psa wiek_psa czyLubiInnePsy opiekun_psa
## 5  Quentin        9          FALSE     Quennie
## 6    Fanta      0.5           TRUE       Filip
```

Pętle oraz instrukcje warunkowe

Pętli oraz instrukcji warunkowych używamy, kiedy chcemy uniknąć powielania kodu i chcemy zachować jego przejrzystość. Ułatwia to wprowadzanie potencjalnych zmian. Instrukcje opisujące co powinno się zdarzyć należy umieścić w nawiasach `{ }`. Jeśli chcemy wykonać tylko jedną linijke kodu, możemy je opuścić.

Instrukcja warunkowa if... else

Umożliwia warunkowe wykonanie kawałka kodu - jeśli warunek zawarty w `if` jest spełniony, to R przejdzie do zawartej instrukcji. W przeciwnym wypadku wykona polecenie zawarte w `else`, a jeśli go nie ma , to przejdzie do kolejnych pętli. Część `else` nie jest wymagana, w tym wypadku z góry wiadomo ile razy kod zostanie wykonany.

Składnia wygląda następująco:

```
if(warunek)
{
  instrukcja_1
}
```

i jest analogiczna do

```
if(warunek) instrukcja_1
```

Możemy także zapisać

```
if(warunek)
{
  instrukcja_1
  instrukcja_2
} else
  {
    instrukcja_3
  }
```

Powiedzmy, że rozpatrujemy liczbe z rozkładu normalnego i sprawdzamy jakiego jest znaku.

```
x_norm <- rnorm(1)

if (x_norm < 0)
{
  cat("Liczba", x_norm, "jest ujemna")
} else
{
  cat("Liczba ", x_norm, "jest dodatnia")
}
```

```
## Liczba  0.04396868 jest dodatnia
```

Możemy chcieć wykonać różne operacje na tak wylosowanej liczbie. Przykładowo, jeśli będzie ujemna, to zmienić znak, zaokrąglić i zreplikować w wektorze

```
if (x_norm < 0)
{
  x_norm <- abs(x_norm)
  x_wek <- rep(round(x_norm, 2), times = 5)
} else
{
  x_wek <- "X"
}
```

i otrzymać X (X oznacza, że wylosowana liczba była dodatnia, a z nią nic nie robimy).

Pętla while

Pętla `while` działa tak długo, dopóki warunek jest spełniony - tzn. do kiedy nie dostaniemy `FALSE`. Warunek należy opisać tak, żeby w pewnym momencie został spełniony - inaczej pętla będzie działać w "nieskończoność". Często używa sie jej do szukania losowych liczb o pewnych właściwościach.

Składnia tej pętli jest następująca:

```
while(warunek)
{
  instrukcja_1
  instrukcja_2
}
```

Tutaj przykład wykorzystania, gdy chcemy losować liczby z przedziału [1, 100], dopóki różnica między dwoma kolejnymi nie będzie parzysta

```
i <- 2
los <- c()
los[1] <- 0
roznica <- 1
while(roznica%%2 != 0)
{
  los <- c(los, sample(1:100, 1, replace = TRUE))
  roznica <- los[i]-los[i-1]
  i = i+1
}
```

W ten sposób dostajemy wylosowane liczby: 0, 9, 24, 58, z różnicą między ostatnimi równą 34.

Pętla for

Pętla `for` wygląda następująco:

```
for(iterator in warunek)
{
  instrukcja_1
  instrukcja_2
}
```

Ta pętla wykonuje instrukcje określoną ilość razy - tyle ile elementów `iterator` w zbiorze `warunek`. W warunku możemy mieć liste albo wektor. Po każdym wykonaniu pętli, zmienna `iterator` przeskakuje do kolejnego elementu warunku.

Jeśli chcemy wykonać tylko 1 instrukcje, można zapisać

```
for(iterator in warunek) instrukcja_1
```

Przykładowo, jeśli chcemy elementy ze zbioru [1, 10] podnieść do potęgi, możemy użyć pętli `for`.

```
wynik <-c()
for (i in 1:10) wynik <- c(wynik, i*i)
wynik
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

Możemy także napisać pętle zagnieżdżone, przykładowo do obliczenia wartości w macierzach. W tym wypadku wartością każdego elementu macierzy (3x3) jest iloczyn jego indeksów, co daje następujący wynik

```
macierz <- matrix(nrow=3, ncol=3)

for(i in 1:dim(macierz)[1])
  {
    for(j in 1:dim(macierz)[2])
      {
        macierz[i,j] = i*j
      }
}

macierz
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    2    4    6
## [3,]    3    6    9
```

Rodzina funkcji apply

Teraz zajmiemy się rodziną funkcji `apply`. Należą do niej takie funkcję jak `apply, tapply, sapply, lapply, vapply`. Wszystkie one pozwalają na wykonanie pewnej operacji na szeregu podzbiorów danych. Operacja, która ma być wykonana określana jest przez argument `FUN`. Funkcje z tej rodziny przyjmują elementy listy (`lapply()`), elementy wektora (`sapply()`), macierze (`apply()`) oraz podgrup wskazanych przez jedną lub kilka zmiennych (`by()` i `tapply()`).

Zacznijmy od funkcji `lapply()`. Wykonuje funkcję `FUN` dla wszystkich elementów wektora `x`. Przydatna funkcja zastępująca pętlę `for`. Domyślnie wynikiem działania jest lista, lecz jeżeli w wyniku chcielibyśmy otrzymać wektor, to jednym z rozwiązań jest zamiana listy na wektor funkcją `unlist()`. Oto przykładowe działanie funkcji `lapply()`:

```
x=c(1,2,3,4,5,6,7,8,9,10)
func=function(x){return(x**3-3*x)}
lapply(x,func)
```

```
## [[1]]
## [1] -2
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 18
##
## [[4]]
## [1] 52
##
## [[5]]
```

```
## [1] 110
##
## [[6]]
## [1] 198
##
## [[7]]
## [1] 322
##
## [[8]]
## [1] 488
##
## [[9]]
## [1] 702
##
## [[10]]
## [1] 970
```

Funkcja `sapply` jest bardziej przyjazną użytkownikowi wersją `lapply` zwracającą wektor lub macierz i może przyjmować więcej argumentów, np. `sapply(x, f, simplify = FALSE, USE.NAMES = FALSE)` zwraca ten sam wynik co `lapply(x, f)`.

Funkcja `vapply` jest podobna do `sapply`, ale ma z góry określony typ zwracanych wartości, a może być również bezpieczniejszy w użyciu, a czasem nawet szybszy.

Teraz weźmiemy pod lupe `tapply()`, która to wykonuje funkcję `FUN` dla podzbiorów wektora `x` określonego przez poziomy zmiennej czynnikowej `index`. Przydatna funkcja, gdy chcemy policzyć pewną statystykę w podgrupach, np. odchylenie standardowe w z wagami. W tym przypadku `x` będzie wektorem z wagami, `index` wektorem z płcią a `FUN` będzie funkcją sd).

```
x=c(98,67,65,82,55,60,72,81,48,88)
index=c('M','M','K','M','K','M','M','M','K','M')
tapply(x,index,sd)
```

```
##         K         M
##  8.544004 12.944938
```

A teraz bardziej zaawansowana werssa funkcji `tapply()` z tą różnicą, że `x` może być macierzą lub listą, `index` może być listą, a wynik tej funkcji jest specyficznie wyświetlany. Jeżeli `index` jest listą zmiennych czynnikowych, to wartość funkcji `FUN` będzie wyznaczona dla każdego przecięcia czynników tych zmiennych. Wynik funkcji `by()` jest klasy `by`, ale po usunięciu informacji o klasie, np. poprzez użycie funkcji `unclass()` otrzymujemy zwykłą macierz. Argument `x` może być listą lub macierzą, dzięki czemu do funkcji `FUN` przekazać można kilka zmiennych – elementów/kolumn listy/macierzy `x`.

```
m1=seq(1:9)
x=c('a','b','c','a','b','c','a','b','c')
by(m1,x,mean)
```

```
## x: a
## [1] 4
## ------------------------------------------------------------
## x: b
## [1] 5
## ------------------------------------------------------------
## x: c
## [1] 6
```

Z kolei `mapply()` to wielowymiarowy odpowiednik funkcji `sapply()`. Argumentami tej funkcji jest funkcja `fun` oraz kilka (dwa lub więcej) wektorów o tej samej długości. Wynikiem jest wektor, w którym na pozycji `i`-tej jest wynik funkcji `fun` wywołanej z `i`-tych elementów wektorów będących argumentami.

```
a=function(x,y){return(x**y)}
mapply(a,x=seq(1,101,by=10),y=seq(1:11))
```

```
##  [1] 1.000000e+00 1.210000e+02 9.261000e+03 9.235210e+05 1.158562e+08
##  [6] 1.759629e+10 3.142743e+12 6.457535e+14 1.500946e+17 3.894161e+19
## [11] 1.115668e+22
```