Statystyka i programowanie w R

Laboratoria

Aleš Kozubík
Uniwersytet Żyliński w Żylinie

Project: Innovative Open Source Courses for Computer Science

31. 5. 2021

Co-funded by the
Erasmus+ Programme
of the European Union

Innovative Open Source Courses for Computer Science

This teaching material was written as one of the outputs of the project
"Innovative Open Source Courses for Computer Science",
funded by the Erasmus+ grant no. 2019-1-PL01-KA203-065564.
The project is coordinated by West Pomeranian University of Technology in Szczecin (Poland)
and is implemented in partnership with Mendel University in Brno (Czech Republic)
and University of Žilina (Slovak Republic).
The project implementation timeline is September 2019 to December 2022.

Innovative Open Source Courses for Computer Science

Project was implemented under the Erasmus+.
Project name: "Innovative Open Source courses for Computer Science curriculum"
Project no.: 2019-1-PL01-KA203-065564
Key Action: KA2 – Cooperation for innovation and the exchange of good practices
Action Type: KA203 – Strategic Partnerships for higher education

Consortium: Zachodniopomorski uniwersytet technologiczny w Szczecinie
Mendelova univerzita v Brně
Žilinská univerzita v Žiline

Erasmus+ Disclaimer: This project has been funded with support from the European Commission. This publication reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Copyright Notice: This content was created by the IOSCS consortium: 2019 – 2022.
The content is Copyrighted and distributed under Creative Commons Attribution-ShareAlike 4.0 International License (CC BY-SA 4.0).

Statystyka i programowanie w R

I. Wprowadzenie do R

Instalacja R

Jest on dostępny za darmo w Comprehensive R Archive Network (w skrócie CRAN)..

Jest on dostępny w Internecie pod adresem https://cran.r-project.org.

Prekompilowane binaria są dostępne dla popularnych platform Linux, Mac OS i Windows.

Możemy wybrać odpowiedni serwer lustrzany, aby pobrać pakiet instalacyjny.

Instalacja pakietów R

Istnieje bogaty zestaw pakietów, które rozszerzają funkcjonalność jądra R.

Pakiety zwiększają wydajność R.

zainstalować pakiety, używamy funkcji `install.packages()`

Pierwsze uruchomienie R

eśli mamy zainstalowany R, możemy sprawdzić jego funkcjonalność.

Prostredie R spustíme jednoducho z príkazového riadku zadaním typing:

```
username@host:~$ R
```

Wyświetlane jest krótkie wprowadzenie, po którym następuje

```
>
```

Ten symbol jest znakiem wiersza poleceń środowiska R.

Opuszczenie środowiska R

Środowisko R jest teraz gotowe do pracy..

Aby wyjść ze środowiska R, wystarczy wpisać

```
> q()
```

R odpowiada pytaniem:

```
Save workspace image? [y/n/c]:
```

Jeśli wybrano `y`, cała historia wykonanych poleceń jest zapisywana w pliku `.Rhistory`, który jest zapisywany w katalogu roboczym.

Obszar roboczy i nawigacja

Wszystkie polecenia są wprowadzane interaktywnie w wierszu poleceń.

Po historii poleceń poruszamy się za pomocą klawiszy kursora, strzałek w górę i w dół.

Dzięki temu możemy wracać do starszych komend bez konieczności pisania ich od nowa. Wystarczy wybrać żądane polecenie i wysłać je ponownie za pomocą klawisza Enter.

Jeśli po opuszczeniu środowiska zapiszemy naszą historię, możemy również powrócić do poleceń z poprzedniej sesji.

Komunikacja z OS

Domyślnym katalogiem roboczym jest katalog, w którym został uruchomiony program R. W tym bieżącym katalogu roboczym R odczytuje i zapisuje pliki i wyniki. Sprawdzamy aktualny katalog roboczy za pomocą funkcji `getwd()`.

Bieżący katalog roboczy może zostać zmieniony przy użyciu funkcji `setwd()`.

Aby wykonać polecenia systemu operacyjnego, należy użyć funkcji `system()`.

twórz nowy katalog używając

```
> system("mkdir new")
```

Szukanie pomocy

Funkcja uzyskiwania pomocy ma zazwyczaj prostą formę `help()` lub jest skracana przez operator `?`.

Jeśli chcemy uzyskać informacje o pakietach rozszerzeń, używamy

```
> help(package="nazwa pakietu")
```

Niektóre pakiety zawierają również próbki kodu, które uruchamiamy za pomocą funkcji `demo()`, na przykład

```
> demo(package="stats")
```

R jak kalkulator

Konsola wiersza poleceń umożliwia interaktywne obliczanie wyników operacji i funkcji

```
> 5+3
[1] 8
```

Jeśli nie widzimy początkowego znaku linii poleceń, może to być spowodowane tym, że nie wpisaliśmy kompletnego polecenia

```
> 5-
+
```

Musimy dokończyć resztę polecenia, a następnie wcisnąć Enter lub anulować polecenie wciskając Esc.

Obiekty

R jest językiem zorientowanym obiektowo

W R wszystko jest obiektem i reprezentuje jakieś dane, które zostały zapisane w pamięci

Obiekty mogą mieć dowolne nazwy, ale należy przestrzegać następujących zasad:

- Nazwa artykułu składa się wyłącznie z dużych lub małych liter, cyfr, podkreśleń i kropek„
- Nazwa artykułu zaczyna się od dużej lub małej litery,
- W języku R rozróżniana jest wielkość liter (to znaczy, że A i a są dwoma różnymi obiektami),
- Nazwa elementu nie może zawierać żadnego z zastrzeżonych słów języka R (możesz zobaczyć ich listę wpisując `help(reserved)`).

Tworzenie obiektu

Nowy obiekt tworzy się w prosty sposób za pomocą operatora przypisania.

Operator przypisania ma dwie możliwe formy: <- lub =.

Zaleca się używanie formy <-, ponieważ forma = może czasami powodować błędy:

```
> log(x=25,base=5)
[1] 2
> x
Error: object 'x' not found
> log(x<-25,base=5)
[1] 2
> x
[1] 25
```

Listowanie i usuwanie obiektów

Listę wszystkich istniejących obiektów otrzymujemy jako wyjście funkcji `ls()`.

Obiekty, które nie będą używane w przyszłości, mogą zostać usunięte z pamięci za pomocą funkcji `rm()`.

Statystyka i programowanie w R

II. Struktury danych w R

Typ danych `numeric`

Typ danych `numeric` reprezentuje rzeczywiste liczby dziesiętne.

Jest to domyślny typ każdego nowego obiektu.

Powstaje ona, gdy do dowolnej zmiennej przypiszemy liczbę rzeczywistą.

Typ dowolnego obiektu jest sprawdzany za pomocą funkcji `class()`.

Typ danych `numeric` – przykład

Przyjrzyjmy się przykładowi.

```
1 > x<-12.35
2 > class(x)
3 [1] "numeryczny"
```

Note

Liczba ta jest reprezentowana jako wektor o długości 1. Znak `[1]` na początku linii oznacza pierwszą pozycję w tym wektorze.

Instalacja R

Wstawienie liczby całkowitej do zmiennej nie zmienia jej typu, ale pozostaje ona `numeric`.

Zobacz przykład

```
1 > z<-100
2 > class(z)
3 [1] "numeric"
```

Możemy również zapytać używając funkcji `is.integer()`.

```
1 > is.integer(z)
2 [1] FALSE
```

Typ danych integer

Aby utworzyć obiekt typu integer, używamy funkcji `as.integer()`.

Przykład

```
> a <- as.integer(12)
> a
[1] 12
> class(a)
[1] "integer"
> is.integer(a)
[1] TRUE
```

Typ danych integer

Alternatywnie, zmienne typu integer mogą być przekazywane jako liczby całkowite zakończone literą L.

Przykład

```
1 > n<-as.integer(10)
2 > class(n)
3 [1] "integer"
4 > n<-10L
5 > class(n)
6 [1] integer
```

Typ danych integer

Co się stanie jeśli wstawimy wartość, która nie jest liczbą całkowitą?

```
1  > as.integer(2.718)
2  [1] 2
3  > as.integer(TRUE)
4  [1] 1
```

Wartość jest zaokrąglana lub przekształcana na liczbę całkowitą.

Typ danych integer

Wyjątkiem są znaki lub ciągi znaków

Nie przekształcone

```
> as.integer("frcka")
[1] NA
Warning message:
NAs introduced by coercion
```

Zmiana typu zmiennej

Przy wszelkich obliczeniach należy pamiętać, że typ zmiennej może się zmieniać.

Przykład

```
> x<-as.integer(20)
> class(x)
[1] "integer"
> x<-x/3+1
> x
[1] 7.666667
> class(x)
[1] "numeric"
```

Typ danych `complex`

Środowisko R pozwala na pracę z liczbami złożonymi.

Wartość złożona jest określona w R przez jednostkę urojoną `i`

Przykład

```
> z<-1+2i
> class(z)
[1] "complex"
```

Typ danych complex

Zauważ, że wartość -1 nie jest typu complex, a zatem

```
1 > sqrt(-1)
2 [1] NaN
3 Warning message:
4 In sqrt(-1) : NaNs produced
```

Musimy określić

```
1 > sqrt(-1+0i)
2 [1] 0+1i
```

Typ danych `complex`

Czy znasz jakieś alternatywne rozwiązanie?

Typ danych `complex`

Czy znasz jakieś alternatywne rozwiązanie?

Używamy funkcji `as.complex()`

Typ danych complex

```
1  > sqrt(as.complex(-1))
2  [1] 0+1i
```

Funkcje `sqrt()` i `as.complex()` muszą być wprowadzone w podanej kolejności.

```
1  > as.complex(sqrt(-1))
2  [1] NaN+0i
3  Warning message:
4  In sqrt(-1) : NaNs produced
```

Typ danych complex

Wprowadzając liczbę złożoną z jednostkową częścią urojoną, musisz również wprowadzić współczynnik.

W przeciwnym razie jednostka urojona jest traktowana jako obiekt.

Przyjrzyjmy się przykładowi

```
1 > a<-1+i
2 Error: object 'i' not found
3 > a<-1+1i
4 > a
5 [1] 1+1i
```

Typ danych `logical`

Może przyjmować dwie wartości logiczne `TRUE` lub `FALSE`.

Często tworzone przez porównywanie zmiennych.

```
1  > x<-10;y<-20
2  > z<-x<y
3  > z
4  [1] TRUE
5  > class(z)
6  [1] "logical"
```

Typ danych `logical`

Zdefiniowane są dla nich wszystkie standardowe operacje logiczne

`&`	Logiczne `AND`
`|`	Logicczne `OR`
`+!+`	Negacja

Typ danych `logical`

Ilustracja

```
1  > a<-TRUE;b<-FALSE
2  > a&b
3  [1] FALSE
4  > a|b
5  [1] TRUE
6  > !a;!b
7  [1] FALSE
8  [1] TRUE
```

Typ danych character

Służy do przechowywania ciągów znaków, ciągi są wprowadzane z użyciem cudzysłowów

```
> x<-"facina"
> class(x)
[1] "character"
#Ale i
> x<-as.character(3.1415926)
> x
[1] "3.1415926"
> class(x)
[1] "character"
```

Typ danych character

Łańcuchy znaków mogą być łączone za pomocą paste().

```
1  > name<-"Donald"
2  > surname<-"Knuth"
3  > paste(name,surname)
4  [1] "Donald␣Knuth"
5  # Jesli chcemy
6  > paste(iname,surname,sep=","))
7  [1] "Donald,Knuth"
```

Typ danych `character`

Jak zaimplementować połączenie bez przerw?

Typ danych `character`

Jak zaimplementować połączenie bez przerw?

Definiujemy separator w funkcji `paste()` jako pusty łańcuch, czyli definiujemy `sep=""`.

Typ danych `character`

Jak zaimplementować połączenie bez przerw?

Definiujemy separator w funkcji `paste()` jako pusty łańcuch, czyli definiujemy `sep=""`.

```
paste(name,surname,sep="")
```

Typ danych `character`

Czasami przydatne jest uzyskanie sformatowanego wyjścia za pomocą funkcji `sprintf()`.

Jego składnia jest taka sama jak w C

Znaczniki formatowania

`s`	character string, `NA` znak values are converted to "NA".
`d,i`	numeryczne wartości.
`o`	integer w notacji ósemkowej.
`x,X`	Liczba całkowita w notacji szesnastkowej przy użyciu tego samego rozmiaru dla `a-f` jak w kodzie.
`f`	Wartość stałoprzecinkowa podwójnej precyzji. Liczba miejsc po przecinku jest określana przez precyzję, domyślna wartość to 6.0.
`e,E`	D Wartość podwójnej precyzji, w notacji wykładniczej, używając tego samego rozmiaru dla e jak w kodzie.

Typ danych character – sformatowane dane wyjściowe

```
> sprintf("%s has %i dogs","John",3)
[1] "John has 3 dogs"
> sprintf("Number pi equals %f",pi)
[1] "Number pi equals 3.141593"
> sprintf("Number pi equals %0.12f",pi)
[1] "Number pi equals 3.141592653590"
> sprintf("10! in exponential %e",factorial(10))
[1] "10! in exponential 3.628800e+06"
sprintf("100 in octal notation %o",100)
[1] "100 in octal notation 144"
> sprintf("1000 in hexadecimal notation %X",1000)
[1] "1000 in hexadecimal notation 3E8"
```

Typ danych `character` – funkcja `sub()`

Jeśli chcemy zastąpić część ciągu znaków innym podłańcuchem, używamy funkcji `sub()`.

Ważne jest, aby uważać na jednoznaczność podłańcucha, ponieważ tylko pierwsze wystąpienie jest zastępowane

Przyjrzyjmy się przykładowi

```
1 > z<-"Here␣is␣my␣brother␣and␣my␣sister"
2 > sub("my","your",z)
3 [1] "Here␣is␣your␣brother␣and␣my␣sister"
4 > sub("my␣sister","your␣sister",z)
5 [1] "Here␣is␣my␣brother␣and␣your␣sister"
```

Funkcja ta różni się od `sub()` tym, że `gsub()` sekwencyjnie zastępuje wszystkie wystąpienia pasującego podłańcucha.

Przyjrzyjmy się zmianie w poprzednim przykładzie

```
1 > gsub("my","your",z)
2 [1] "Here␣is␣your␣brother␣and␣your␣sister"
```

Wektory

Wektor jest najprostszą strukturą danych.

Można ją scharakteryzować jako dostępność elementów tego samego typu danych.

Poszczególne wartości zawarte w wektorze są określane jako składowe.

Liczba składowych wektora jest określana jako jego długość.

Wektory

Wektor `v` jest tworzony za pomocą funkcji `c()`.

Jego długość jest znajdowana przy użyciu funkcji `length()`.

```
1  > v<-c(1,3,5,7,9)
2  > length(v)
3  [1] 5
```

Wektory

Wektor wartości logicznych

```
1 > v<-c(TRUE,TRUE,FALSE,TRUE,FALSE)
2 >v
3 [1] TRUE TRUE FALSE FALSE TRUE FALSE
```

Wektor elementów typu character

```
1 > a<-c("aa","bb","cc","dd","ee","ff")
2 > a
3 [1] "aa" "bb" "cc" "dd" "ee" "ff"
```

Wektory

Wektory mogą być łączone przy użyciu “combine” `c()`.

```
1  >a<-c(1,2,3)
2  >b<-c(4,5,6)
3  >c(b,a)
4  [1] 4 5 6 1 2 3
5  # Patrz nadpisywanie komponentow
6  > a<-c("a", "b", "c")
7  > c(a,b)
8  [1] "a" "b" "c" "4" "5" "6"
```

Wektory – arytmetyka

Arytmetyka wektorowa jest implementowana komponent po komponencie.

Operacje arytmetyczne są implementowane komponent po komponencie.

- `+` dodawanie liczby do wszystkich składowych lub dodawanie wektorów po składowych
- `-` odejmuje liczbę od wszystkich składowych lub odejmuje wektory składowa po składowej,
- `*` pomnożyć wszystkie składowe przez liczbę lub pomnożyć wektory przez składowe,
- `/` dzielenie wszystkich składowych przez liczbę lub dzielenie wektorów przez składowe.

Wektory – arytmetyka

```
> v<-c(1,3,5,7,9)
> u<-c(10,20,30,40,50)
> u+v
[1] 11 23 35 47 59
> u-v
[1] 9 17 25 33 41
> 5*v
[1] 5 15 25 35 45
> u*v
[1] 10 60 150 280 450
> u/5
[1] 2 4 6 8 10
> u/v
[1] 10.000000 6.666667 6.000000 5.714286 5.555556
```

Wektory – arytmetyka

Ostrzeżenie o zasadach recyklingu

Jeśli długości wektorów nie są zgodne, to krótszy z nich jest używany cyklicznie wielokrotnie.

Ta reguła jest ograniczona przez warunek, że długość dłuższego wektora jest integralną wielokrotnością krótszego. Jeśli nie, operacja nie jest wykonywana.

Wektory – arytmetyka

```
1  > v<-c(10,20,30)
2  > u<-1:9
3  > u+v
4  [1] 11 22 33 14 25 36 17 28 39
5  # Liczba jest wektorem o dlugosci 1
6  > b<-c(1,2,3,4)
7  > 5*b
8  [1] 5 10 15 20
```

Wektory – wybór komponentów

Składowe, które chcemy wybrać z wektora, określają indeksy w nawiasach `[]`.

```
1  > v<-1:10
2  > v[3:5]
3  [1] 3 4 5
```

Note

Operator `:` definiuje zakres liczb od pierwszej do drugiej.

Wektory – wybór komponentów

Komponenty mogą być również wybierane za pomocą wektora wartości logicznych.

Długość obu wektorów powinna być taka sama, w przeciwnym razie przyjmuje się, że pozostałe pozycje są Ž00+TRUE+.

```
1  > u<-2*1:6
2  > L<-c(FALSE,TRUE,TRUE,FALSE,FALSE,TRUE)
3  > u[L]
4  [1]  4  6  12
```

Wektory – wybór komponentów

Wybrane elementy nie muszą znajdować się w ciągłej sekwencji indeksów.

Definiuje się je za pomocą funkcji `c()`.

```
1  > a<-c("aa","bb","cc","dd","ee","ff")
2  > a[c(2,3,5)]
3  [1] "bb" "cc" "ee"
4  # Powtarzajace sie indeksy
5  > a[c(2,2,3,5)]
6  [1] "bb" "bb" "cc" "ee"
```

Wektory - nadawanie nazw składnikom

Komponentom możemy nadać odpowiednie nazwy.

Nazwy definiujemy za pomocą funkcji `names()`.

```
1  > v<-c("Donald","Knuth")
2  > names(v)<-c("Name","Surname")
3  > v
4       Name   Surname
5  "Donald"   "Knuth"
```

Wektory - nadawanie nazw składnikom

Po przypisaniu nazw do komponentów, możemy je wybrać używając tych nazw.

W poprzednim przykładzie, możemy użyć

```
1 > v["Surname"]
2 Surname
3 "Knuth"
```

Macierz

Macierz to dwuwymiarowa tablica danych tego samego typu ułożonych w prostokątny schemat.

Tworzymy ją za pomocą funkcji `matrix()` z następującymi argumentami

`vector`	zawiera elementy macierzy,
`nrow`	jest wartością całkowitą, która wskazuje liczbę wierszy w mactierzy,
`ncol`	jest wartością całkowitą określającą liczbę kolumn macirzy,
`byrow`	jest wartością logiczną, która określa, czy macierz powinna być wypełniona wierszami (`byrows=TRUE`) czy kolumnami (`byrows=FALSE`), jej domyślną wartością jest `FALSE`,
`dimnames`	jest listą wektorów typu `character`, które zawierają opcjonalne etykiety wierszy i kolumn.

Macierz – przykładowe zadanie

vfill

```
> A<-matrix(3:8,nrow=3,ncol=2,byrow=TRUE)
> A
     [,1] [,2]
[1,]    3    4
[2,]    5    6
[3,]    7    8
> B<-matrix(3:8,nrow=3,ncol=2,byrow=FALSE)
> B
     [,1] [,2]
[1,]    3    6
[2,]    4    7
[3,]    5    8
```

Macierz – dostęp do elementów

Dostęp do poszczególnych elementów macierzy uzyskuje się za pomocą pary indeksów oddzielonych przecinkami w nawiasach.

```
1  > A[2,2]
2  [1] 6
```

Macierz – dostęp do elementów

Dostęp do poszczególnych elementów macierzy uzyskuje się za pomocą pary indeksów oddzielonych przecinkami w nawiasach.

```
1  > A[2,2]
2  [1] 6
```

Pominięcie jednego z indeksów prowadzi do wyodrębnienia wiersza lub kolumny.

```
1  > A[,1]
2  [1] 3 5 7
3  > B[2,]
4  [1] 4 7
```

Macierz – wybór podmacierzy

Wiersze i kolumny definiuje się za pomocą funkcji `c()`.

```
> C<-matrix(1:12,nrow=3)
> C
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> C[c(1,3),c(2,4)]
     [,1] [,2]
[1,]    4   10
[2,]    6   12
```

Macierz – przypisanie nazwy

Nazwy przypisujemy do wierszy i kolumn za pomocą funkcji `dimnames()` i `list()`.

```
> dimnames(A)<-list(c("row1","row2","row3"),
+ c("col1","col2"))
> A
     col1 col2
row1    3    4
row2    5    6
row3    7    8

> A["row2","col1"]
[1] 5
```

Macierz – transpozycja

Możemy transponować macierz za pomocą funkcji `t()`.

```
> B<-matrix(3:8,nrow=3,ncol=2,byrow=FALSE)
> t(B)
     [,1] [,2] [,3]
[1,]    3    4    5
[2,]    6    7    8
```

Pozostałe funkcje są zdefiniowane w pakiecie `matlib`.

Macierz – operacje

Są one określane dla poszczególnych komponentów

Jest to ważne w mnożeniu macierzy. Wspólna operacja erb+*+ oznacza mnożenie elementów na tych samych pozycjach.

Standardowe mnożenie macierzy z algebry liniowej jest definiowane jako operacja %*%.

Macierz – operacje

Porównaj

```
1  > C<-B[c(1,2),c(1,2)]
2  > C*C
3        [,1] [,2]
4  [1,] 9 36
5  [2,] 16 49
6  > C%*%C
7        [,1] [,2]
8  [1,] 33 60
9  [2,] 40 73
```

Macierz – łączenie

Aby połączyć macierze, muszą one mieć taką samą liczbę wierszy lub kolumn.

Jeśli mają taką samą liczbę wierszy, możemy połączyć kolumny używając funkcji `cbind()`.

```
1  > cbind(B,diag(c(1,2,5))
2       [,1] [,2] [,3] [,4] [,5]
3  [1,]  3  6  1  0  0
4  [2,]  4  7  0  2  0
5  [3,]  5  8  0  0  5
```

Uwaga

Zwróć uwagę na funkcję `diag()`. Tworzy macierz diagonalną z podanym wektorem na przekątnej.

Macierz – łączenie

Jeśli macierze mają taką samą liczbę kolumn, możemy je połączyć za pomocą funkcji `rbind()`.

```
> C<-matrix(1:12,nrow=3)
> rbind(C,diag(c(1,2,5,7))[c(2,4),])
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
[4,]    0    2    0    0
[5,]    0    0    0    7
```

Array

Tablice są uogólnieniem macierzowej struktury danych.

W rzeczywistości są to macierze więcej niż dwuwymiarowe.

Tablice można tworzyć za pomocą funkcji `array()`.

Składnia tej funkcji jest następująca

```
name<-array(vector, dimensions,dimnames)
```

Pole – tworzenie

Utwórzmy tablicę o wymiarach $3 \times 4 \times 3$.

Aby lepiej poruszać się po tablicy, stwórzmy najpierw nazwy poszczególnych wymiarów.

```
1 > dim1<-c("A1", "A2", "A3")
2 > dim2<-c("B1", "B2", "B3", "B4")
3 > dim3<-c("C1", "C2", "C3")
```

Pole - tworzenie, kontynuacja

Teraz tworzymy tablicę, która zawiera liczby całkowite od 1 do 36 ($3\times$ 4×3 to 36).

```
> z<-array(1:36,c(3,4,3),
dimnames=list(dim1,dim2,dim3))
```

Aby wyświetlić strukturę tablicy, wpisz w R: z.

Dane wyjściowe są zbyt długie, aby można je było wyświetlić w prezentacji.

Pole – dostęp do elementów

Dostęp do elementów tablicy odbywa się za pomocą nawiasów kwadratowych w taki sam sposób jak do macierz.

```
> z[2,3,1]
[1] 8
> z[2:3,2:3,2]
    B2 B3
A2 17 20
A3 18 21
```

Struktura data frame

Data frame jest najbardziej popularną strukturą do przechowywania danych.

Umożliwia przechowywanie wektorów kolumnowych różnych typów danych.

Data framy tworzone są za pomocą funkcji `data.frame()`, której ogólna składnia jest następująca

```
1 > name<-data.frame(col1,col2,col3, ...)
```

Data frame – tworzenie

.

Utwórz krótką ramkę danych zawierającą dane o strzałach koszykarzy.

```
> playerID<-c(1,2,3,4)
> position<-c("forward","guard","forward","center")
> attempted<-c(12,6,10,15)
> made<-c(7,4,6,12)
> players<-data.frame(playerID,position,attempted,made)
> players
  playerID position attempted made
1        1  forward        12    7
2        2    guard         6    4
3        3  forward        10    6
4        4   center        15   12
```

Data frame – dostęp do komórek

Dostęp do poszczególnych komórek ramki danych można uzyskać na wiele sposobów.

Używanie indeksów

```
> players[1:2]
  pposition playerID
1 1 rorward
2 2 guard
3 3 forward
4 4 center
```

Data frame – dostęp do komórek

Inną opcją jest użycie nazw kolumn.

Nazwy kolumn są określone jako wektor typu `character`.

```
1  > players[c("playerID", "attempted", "made")]
2    playerID attempted made
3  1 1 12 7
4  2 2 6 4
5  3 3 10 6
6  4 4 15 12
```

Data frame – dostęp do komórek

Trzecią opcją jest użycie tagowania.

Składa się ona z nazwy ramki danych na pierwszej pozycji i nazwy kolumny na drugiej pozycji, oddzielonych znakiem $.

```
1 > players$position
2 [1] forward guard   forward center
3 Levels: center forward guard
```

Data frame – operator podwójnego nawiasu

.

Aby uzyskać dostęp do pojedynczej kolumny, użyj podwójnych nawiasów kwadratowych.

Porównaj te dwa stwierdzenia

```
> players[4]
  made
1    7
2    4
3    6
4   12
```

```
> players[[4]]
[1]  7  4  6 12
```

Data frame – operator podwójnego nawiasu

Operator podwójnego nawiasu jest równoważny użyciu przecinka w operatorze pojedynczego nawiasu.

```
> players[,4]
[1] 7 4 6 12
> players[, "made"]
[1] 7 4 6 12
```

Data frame – przypisywanie nazw do wierszy

Używamy funkcji `row.names()`, której argumentem jest wektor typu `character`.

```
> row.names(players)<-c("Player1","Player2","Player3","Player4")
> players
         playerID position attempted made
Player1         1  forward        12    7
Player2         2    guard         6    4
Player3         3  forward        10    6
Player4         4   center        15   12
```

Data frame – przypisywanie nazw do wierszy

Teraz możemy wyodrębnić wiersze według indeksu lub według nazwy.

```
> players[3,]
         playerID position attempted made
Player3        3  forward        10    6
> players["Player3",]
         playerID position attempted made
Player3        3  forward        10    6
```

Data frame – wybór wiersza

.

Jeśli chcemy uzyskać więcej niż jeden wiersz, używamy wektora liczb całkowitych.

```
> players[c(1,3),]
         playerID position attempted made
Player1         1  forward        12    7
Player3         3  forward        10    6
> players[2:4,]
         playerID position attempted made
Player2         2    guard         6    4
Player3         3  forward        10    6
Player4         4   center        15   12
```

Data frame – przypisywanie nazw do wierszy

Jeśli musimy często kopiować nazwę ramki danych, może to być niewygodne.

Użyj `attach()`, aby dodać ramkę danych do ścieżki wyszukiwania.

Dzięki temu możemy pisać tylko nazwy kolumn.

Jeśli chcemy usunąć ramkę danych ze ścieżki wyszukiwania, po prostu używamy funkcji `detach()`.

Data frame – attach() próbka

Po połączeniu ramki danych players możemy łatwo obliczyć procenty każdego gracza.

```
> attach(players)
The following objects are masked _by_ .GlobalEnv:

    attempted, made, playerID, position

> 100*made/attempted
[1] 58.33333 66.66667 60.00000 80.00000
```

Data frame – alternatywa dla `attach()`

Alternatywą dla dołączenia ramki danych do ścieżki wyszukiwania jest użycie funkcji `with()`.

```
> with(players, {
+ 100*made/attempted}
+ )
[1] 58.33333 66.66667 60.00000 80.00000
```

Data frame – łączenie

.

Często zachodzi potrzeba połączenia danych z dwóch lub więcej zbiorów danych.

Używamy funkcji `merge()`.

Argumenty są nazwami dwóch ramek danych, które mają zostać połączone.

Trzeci argument, `by=''column_name''`, określa zmienną, która ma zostać połączona z danymi.

Data frame – łączenie danych

Aby zademonstrować łączenie, najpierw tworzymy nową ramkę danych `rebounds`.

```
1 > offensive<-c(5,2,3,10)
2 > defensive<-c(6,3,8,12)
3 > rebounds<-data.frame(playerID,defensive,offensive)
4 > row.names(rebounds)<-c("Player1","Player2","Player3",
5 + "Player4")
```

Data frame – łączenie danych

Teraz jesteśmy gotowi do połączenia ramek danych `players` i `rebounds`.

```
> new_players<-merge(players,rebounds,by="playerID")
> new_players
  playerID position attempted made defensive offensive
1        1  forward        12    7         6         5
2        2    guard         6    4         3         2
3        3  forward        10    6         8         3
4        4   center        15   12        12        10
```

Data frame – łączenie danych

.

Alternatywą jest dodanie wierszy do istniejącej ramki danych za pomocą funkcji `rbind()`.

Argumentami są nazwy dwóch ramek danych.

Aby to zilustrować, przygotujemy nową ramkę danych `players2`

```
1 > position<-c("center","guard","forward")
2 > attempted<-c(14,8,12)
3 > made<-c(10,5,8)
4 > players2<-data.frame(playerID,made,attempted,position)
5 > row.names(players2)<-c("Player5","Player6","Player7")
```

Data frame – łączenie danych

Teraz połączymy te ramki danych.

```
> more_players<-rbind(players,players2)
> more_players
         playerID position attempted made
Player1         1  forward        12    7
Player2         2    guard         6    4
Player3         3  forward        10    6
Player4         4   center        15   12
Player5         5   center        14   10
Player6         6    guard         8    5
Player7         7  forward        12    8
```

Listy- `List`

Listy są najbardziej złożoną strukturą danych.

Reprezentują one uporządkowane kolekcje obiektów.

Aby utworzyć listę, użyj funkcji `list()`. Jego składnia jest prosta:

```
\list(object1,object2,...)
```

Jego argumentami są nazwy istniejących obiektów

Listy

Opcjonalnie nadaj nazwy obiektom na tworzonej liście:

```
\list(name1=object1,name2=object2,...)
```

Listy

Z naszych istniejących data framow players i players2 utworzymy listę o nazwie NBA.

```
> NBA<-list(club="Bulls",city="Chicago",Players=players)
> NBA
$club
[1] "Bulls"

$city
[1] "Chicago"

$Players
         playerID position attempted made
Player1         1  forward        12    7
Player2         2    guard         6    4
Player3         3  forward        10    6
Player4         4   center        15   12
```

Listy

Teraz możemy dodać kolejny element listy używając funkcji konkatenacji `c()`.

```
1 > NBA<-c(NBA,list(club="Celtics",city="Boston",Players=players2))
2 > NBA
```

Dane wyjściowe są zbyt długie, aby je tutaj pokazać, zobacz je bezpośrednio w R.

Uwaga

Ta funkcja łączy wszystkie argumenty w jedną strukturę wektorową. W tym przypadku oznacza to, że drugi klub otrzymał pozycje od 4 do 6 na nowej liście, natomiast element podwójnie indeksowany `[2,1]` nie istnieje na liście.

Listy – dostęp do elementów

Konieczne jest rozróżnienie pomiędzy operatorami pojedynczego i podwójnego nawiasu.

Spróbuj następujących poleceń (niektóre dane wyjściowe są zbyt długie, aby je tu pokazać)

```
> NBA[3]
> NBA[[3]]
> NBA[3][2]
$<NA>
NULL
> NBA[[3]][2,]
         playerID position attempted made
Player2         2    guard         6    4
```

Listy – edycja elementu

Notacja podwójnego nawiasu pozwala na bezpośrednią modyfikację elementów listy.

```
> NBA[[3]][2,]
         playerID position attempted made
Player2         2    guard         6    4
> NBA[[3]][2,3]<-c(7)
> NBA[[3]][2,]
         playerID position attempted made
Player2         2    guard         7    4
```

Wejście z klawiatury

Najprostsza metoda (ale również najbardziej czasochłonna dla dużych próbek)

Pracujemy w dwóch etapach

- Utwórz pustą ramkę danych z nazwami i typami zmiennych, które chcemy przechowywać.
- Otwórz prosty edytor danych za pomocą funkcji *()+, której argumentem jest nazwa ramki danych, którą chcemy edytować.*

Wejście z klawiatury

Tworzymy pustą ramkę danych o nazwie `mydata` z czterema zmiennymi: `name`, która ma typ `character`, oraz trzy zmienne numeryczne age, `height` i `weight`.

```
1 > mydata<-data.frame(name=character(0),age=numeric(0),
2 + height=numeric(0),weight=numeric(0))
3 > mydata<-edit(mydata)
```

Uwaga

Zauważ, że przypisania takie jak `numeric(0)` i `character(0)` utworzą zmienną tego typu, ale bez danych.

Dane wejściowe z pliku .csv

Wartości rozdzielone przecinkiem, jeden z najczęściej używanych formatów danych.

Pierwszy wiersz może, ale nie musi zawierać nazw kolumn.

Przykład struktury plików

```
Column1,Column2,Column3
A,10,0.11
B,20,0.22
C,30,0.33
```

Dane wejściowe z pliku .csv

Zakładamy, że dane są przechowywane w pliku `mydata.csv`.

Importujemy dane za pomocą funkcji `read.csv()`.

```
> mydata<-read.csv("mydata.csv")
> class(mydata)
[1] "data.frame"
> mydata
  Column1 Column2 Column3
1       A      10    0.11
2       B      20    0.22
3       C      30    0.33
```

Dane wejściowe z pliku .csv

Opcjonalne argumenty do `read.csv()`.

- `header` Wartość logiczna, określa czy plik wejściowy zawiera nazwy zmiennych jako pierwszą linię, wartość domyślna `TRUE`.
- `sep` określa znak oddzielający wpisy, domyślną wartością jest przecinek,
- `dec` określa znak używany w pliku dla dziesiętnych, domyślną wartością jest `.`, należy również wspomnieć o funkcji `read.csv2()`, która używa przecinka dla dziesiętnych i średnika jako separatora.
- `skip=n` określa liczbę linii do pominięcia przed odczytem danych. Opcja ta jest przydatna dla tabel danych z pustymi wierszami lub opisów tekstowych na początku plików.
- `stringsAsFactors`, która jest wartością logiczną określającą czy ciągi są konwertowane na czynniki, ustaw ją na `FALSE` jeśli chcesz zapobiec konwersji.
- `row.names` jest wektorem nazw wierszy.

Zapis danych do pliku `.csv`

R może utworzyć plik `csv` z istniejących data framu.

Używamy funkcji `write.csv()` lub `write.csv2()`, która używa przecinka jako kropki dziesiętnej i średnika jako separatora.

Wspólna składnia

```
write.csv(object,file="filename",...options)
```

`object` jest obowiązkowym argumentem zawierającym nazwę data framu, którą chcemy zapisać, a `filename` jest nazwą (lub pełną ścieżką) pliku.

Zapisywanie danych do pliku `.csv`

Wybrane opcje programu `write.csv()`.

- `append`, który jest wartością logiczną wskazującą, czy dane wyjściowe są dołączane do końca pliku. Domyślną wartością jest `FALSE` i wszystkie istniejące pliki o tej nazwie zostaną nadpisane.
- `sep` Znak separatora elementów jest definiowany przez czasownik `sep`. Wartości w każdej linii `object` są oddzielone tym znakiem.
- `dec` łańcuch, który ma być używany dla punktów dziesiętnych w kolumnach numerycznych lub złożonych, musi to być pojedynczy znak. Domyślną wartością jest kropka dziesiętna.
- `row.names` Wartość logiczna określająca, czy zapisywać nazwy wierszy`object`.

Dane wejściowe z plików Excela

Istnieje kilka pakietów, które pozwalają na import danych bezpośrednio z plików Excela. Przyjrzyjmy się niektórym z nich:

- `xlsx`,
- `XLconnect`
- `readxl`

Excel 2007 i późniejsze wersje używają formatu `xlsx`, więc tutaj wymienimy pakiet `xlsx`.

Dane wejściowe z plików Excela

Instalujemy pakiet za pomocą zwykłej komendy:

```
install.packages("xlsx")
```

Jeśli chcemy go użyć w bieżącym obszarze roboczym, ładujemy go w standardowy sposób:

```
library("xlsx")
```

Dane wejściowe z plików Excela

Pakiet ten dostarcza dwie funkcje do ładowania zawartości skoroszytu Excela do R data.frame: `read.xlsx()` i `read.xlsx2()`.

Różnica pomiędzy tymi dwoma funkcjami jest następująca:

- `read.xlsx()` zachowuje typ danych, typ zmiennej odpowiada każdej kolumnie w arkuszu, ale jest powolny dla dużych zestawów danych (arkusz z więcej niż 100 000 komórek). `read.xlsx2()` jest szybszy dla dużych plików.

Dane wejściowe z plików Excela

Obie funkcje mają podobną składnię:

```
read.xlsx(file, sheetIndex, header=TRUE, colClasses=NA)
read.xlsx2(file, sheetIndex, header=TRUE, colClasses="character")
```

Ich argumenty mają następujące znaczenie:

- `file` to nazwa pliku, który zawiera tabelę. Jeżeli plik nie znajduje się w katalogu roboczym, musi być podana pełna ścieżka.
- `sheetIndex` jest liczbą wskazującą indeks arkusza, który ma zostać odczytany. Można go zastąpić argumentem `sheetname`, podanym jako łańcuch znaków z nazwą arkusza.
- `header` wartość logiczna. Jeśli `header=TRUE`, to pierwsza linia jest używana jako konwencja nazewnictwa zmiennych.
- `colClasses` wektor znaków reprezentujący klasę każdej kolumny.
- `startRow`,liczby określające indeks początkowego i ostatniego załadowanego wiersza.

Zapisywanie danych do plików Excela

Pakiet `xlsx` udostępnia dwie funkcje zapisu `write.xlsx()` oraz `write.xlsx2()`.

Ogólna składnia

```
write.xlsx(x, file, sheetName="Sheet1", col.names=TRUE,
row.names=TRUE, append=FALSE)

write.xlsx2(x, file, sheetName="Sheet1",col.names=TRUE,
row.names=TRUE, append=FALSE)
```

Zapisywanie danych do plików Excela

Ich argumenty mają następujące znaczenie:

- `x` data frame, która jest zapisywana do skoroszytu.
- `file` nazwa (lub ścieżka) pliku wyjściowego.
- `sheetName` łańcuch znaków z nazwą arkusza.
- `col.names` Wartość logiczna określająca, czy zapisywać nazwy kolumn do pliku.
- `row.names` Wartość logiczna, określa czy zapisywać nazwy wierszy do pliku.
- `append` Wartość logiczna, określa czy `x` powinien być dołączony do istniejącego pliku, jeśli `FALSE`, nadpisuje istniejący plik w ten sam sposób.

Odczytywanie danych z plików JSON

JSON (JavaScript Object Notation) jest prostym formatem wymiany danych.

Aby móc czytać pliki JSON w R, musimy najpierw zainstalować lub załadować pakiet `rjson`.

Możemy użyć funkcji `fromJSON()`.

Sposób użycia zależy od lokalizacji pliku `.json`.

```
dane<-fromJSON(file="filename.json")
dane<-fromJSON(file="URL do pliku json")
```

W obu przypadkach, obiekt `data` jest przechowywany jako lista. Do dalszej analizy możemy przekonwertować dane za pomocą funkcji `as.data.frame()`.

Zapisywanie danych do plików JSON

Musi być wykonany w dwóch etapach.

W pierwszym kroku musimy przygotować obiekt JSON, a w drugim zapisać go do pliku.

Aby utworzyć obiekt JSON, używamy funkcji `toJSON()`:

```
dataJSON<-toJSON(dane)
```

Następnie używamy funkcji `write()`.

```
write(dataJSON, "filename.json")
```

Statystyka i programowanie w R

III. Rozkłady prawdopodobieństwa w R

Wybór losowy

Funkcja standardowa `sample()` o składni

```
sample(x, size, replace, prob)
```

Argumenty

- `x` to wektor lub zbiór danych, z którego wybierana jest próbka,
- `size` Wielkość próby
- `replace` jest wartością logiczną, która określa, czy wartości w próbce są powtarzane, czy nie,
- `prob`

Wybór losowy – przykłady

Najprościej jest użyć tylko pierwszego argumentu

```
> sample(6)
[1] 4 3 5 1 6 2
> sample(4:10)
[1] 9 7 5 4 8 10 6
> sample(c(1,3,5,7,9))
[1] 9 5 7 3 1
```

Wybór losowy – przykłady

Drugi argument określa liczebność próby

5 losowo wybranych liczb całkowitych od 1 do 40

```
1  > sample(1:40,5)
2  [1] 30 35 34 5 29
```

Wybór losowy – przykłady

Symulacja 50-krotnego rzutu kostką

```
1  > sample(6,50)
2  Error in sample.int(x, size, replace, prob) :
3   cannot take a sample larger than the population when 'replace = FALSE'
```

Błąd, ponieważ rozmiar próbki przekracza długość wektora danych, z którego ma być pobrana próbka.

Wybór losowy – przykłady

Symulujemy pięćdziesięciokrotny rzut kostką - aby powtórzyć wartości, musimy ustawić argument `replace`.

```
> sample (6 , 50 , replace = TRUE )
 [1] 6 5 3 3 5 5 4 6 3 1 3 2 ...
[39] 2 6 6 6 4 2 2 5 1 6 1 5
```

Wybór losowy – przykłady

Możemy również zasymulować rzucanie fałszywą monetą z większą częstotliwością główek.

Załóżmy, że kiery pojawiają się dwa razy częściej niż reszki, i ustaw argument `prob=c(2/3,1/3)`.

```
1  > sample (c("head","tail"),20,replace = TRUE,prob=c(2/3,1/3))
2   [1] "head" "tail" "head" "head" "head" "head" "tail" "head" "head" "tail"
3  [11] "head" "tail" "head" "head" "tail" "head" "head" "tail" "tail" "head"
```

Wybory losowe - zapewnienie tego samego wyniku

Jeśli weźmiemy próbki, będą one losowe i będą się zmieniać za każdym razem, gdy użyjemy funkcji `sample()`.

Jeśli potrzebujemy zrekonstruować tę samą próbkę, możemy użyć funkcji `set.seed()`.

```
1 > set.seed(3)
2 > sample(6)
3 [1] 2 5 6 1 4 3
4 > set.seed(3)
5 > sample(6)
6 [1] 2 5 6 1 4 3
```

Rozkład dyskretny

Prawdopodobieństwa są określane przez listę prawdopodobieństw dyskretnych wyników, zwaną funkcją prawdopodobieństwa.

Jeżeli zbiór wszystkich możliwych wartości dyskretnej zmiennej losowej X oznaczymy jako H , to możemy wprowadzić funkcję prawdopodobieństwa $p(x)$ według wzoru

$$p(x) = \mathbb{P}(X = x), \; x \in H. \tag{1}$$

Rozkład dyskretny

Wymieńmy niektóre z nich:

- rozkład Bernoulliego,
- rozkład dwumianowy,
- rozkład geometryczny pozycji,
- rozkład hipergeometryczny,
- rozkład dwumianowy ujemny,
- rozkład Poissona.

Rozkład Bernoulliego

Mamy cztery funkcje:

- `rbern(n,prob)`, gdzie Ž200+ to liczba obserwacji, a `prob` to prawdopodobieństwo zdarzenia losowego A (sukcesu w eksperymencie). Generuje on wektor `0` oraz `1` wybrany z rozkładu Bernoulliego z zadanym prawdopodobieństwem.
- `pbern(q, prob, lower.tail = TRUE, log.p = FALSE)`
- `dbern(x, prob, log = FALSE)`
- `qbern(p, prob, lower.tail = TRUE, log.p = FALSE)`

Rozkład dwumianowy

W programie zaimplementowane są 4 funkcje do pracy z rozkładem dwumianowym:

- `rbinom(n,prob)`, gdzie `n` to liczba obserwacji, `p` to prawdopodobieństwo sukcesu. Funkcja ta generuje n zmiennych losowych z zadanym prawdopodobieństwem.
- `pbinom(x,n k)`, gdzie `n` jest całkowitą liczbą prób, `p` jest prawdopodobieństwem sukcesu, `x` jest wartością, dla której ma być wyznaczone prawdopodobieństwo.
- `dbinom(x,n,p)`, gdzie `n` jest całkowitą liczbą prób, `p` jest prawdopodobieństwem sukcesu, `x` jest wartością, dla której ma być wyznaczone prawdopodobieństwo.
- `qbinom(prob,n,p)`, gdzie `prob` jest prawdopodobieństwem, `n` jest całkowitą liczbą prób, a `p` jest prawdopodobieństwem sukcesu na próbę. Funkcja ta służy do wyznaczania kwantyla n-tego, tzn. wyznacza k takie, że $P(X \leq k)$.

Rozkład dwumianowy – przykłady

Príklad.
Załóżmy, że w teście jest dwadzieścia pytań wielokrotnego wyboru. Każde pytanie ma pięć możliwych odpowiedzi i tylko jedna z nich jest poprawna. Wyznaczyć prawdopodobieństwo co najwyżej sześciu poprawnych odpowiedzi, jeśli student próbuje odpowiedzieć na każde pytanie w sposób losowy.

Rozkład dwumianowy – rozwiązanie 1

Prawdopodobieństwo poprawnej, losowej odpowiedzi na pytanie wynosi $\frac{1}{5} = 0,2$.

Prawdopodobieństwo wystąpienia dokładnie 6 poprawnych odpowiedzi można wyznaczyć za pomocą funkcji `dbinom()`.

```
1 > dbinom(6,20,0.2)
2 [1] 0.1090997
```

Rozkład dwumianowy – rozwiązanie 1

Użyj funkcji`dbinom()` dla $x = 0, 6$, aby znaleźć prawdopodobieństwo wystąpienia sześciu lub mniej poprawnych odpowiedzi w losowych eksperymentach i zsumuj wyniki.

W ten sposób otrzymujemy:

```
1 > dbinom(0,20,0.2) + dbinom(1,20,0.2) + dbinom(2,20,0.2)+
2 + dbinom(3,20,0.2) + dbinom(4,20,0.2) + dbinom(5,20,0.2)+
3 + dbinomial(6,20,0.2)
4 [1] 0.9133075
```

Rozkład dwumianowy – rozwiązanie 2

Alternatywnie możemy użyć funkcji dystrybuanty dla rozkładu dwumianowego `pbinom()`.

Otrzymujemy tę samą wartość

```
1 > pbinom(6,20,0.2)
2 [1] 0.9133075
```

Rozkład dwumianowy – kontynuacja przykładu

Príklad.
Student zdaje egzamin, jeśli odpowie poprawnie na więcej niż 10 pytań na teście. Jakie jest prawdopodobieństwo, że student zda egzamin, jeśli będzie odpowiadał na pytania losowo?

Rozkład dwumianowy – rozwiązanie

Ponieważ szukamy prawdopodobieństwa $\mathbb{P}(X > 10)$. W tym przypadku używamy funkcji `pbinom()`, ale z opcją `lower.tail=FALSE`.

To daje nam

```
1 > pbinom(10,20,0.2,lower.tail=FALSE)
2 [1] 0.0005634137
```

Rozkład dwumianowy – przykład 2

Príklad.
Załóżmy, że jesteśmy odpowiedzialni za jakość w fabryce. Produkujemy 250 urządzeń dziennie. Uszkodzone urządzenie musi zostać naprawione. Wiemy, że współczynnik wadliwości wynosi 2%. Przeprowadźmy symulację ile urządzeń musimy naprawić każdego dnia w tym tygodniu.

Rozkład dwumianowy – rozwiązanie

Za pomocą funkcji `rbinom()` wygeneruj próbę losową z rozkładu dwumianowego o liczbie prób $n = 250$ i prawdopodobieństwie sukcesu $p = 0,02$.

Mamy więc

```
1 > rbinom(7,250,0.02)
2 [1] 2 5 3 9 5 9 5
```

Rozkład dwumianowy – przykład 3

Príklad.
Załóżmy, że testujemy lek, który ma 80% skuteczności. W każdym badaniu bierze udział 30 pacjentów. Ilu pacjentów znajduje się w dolnych 10 procentach odsetka pozytywnych wyników? Wymieńmy decyle w tym teście leczenia.

Rozkład dwumianowy – rozwiązanie

10% udanych prób będzie miało od 0 do 21 pacjentów z pozytywną odpowiedzią na to leczenie. Wyznaczamy to za pomocą funkcji `qbinom()`:

```
1  > qbinom(0,1,30,0,8)
2  [1] 21
```

Aby uzyskać każdy decyl w tym teście przetwarzania, wprowadzamy

```
1  > qbinom(seq(0.1,1,0.1),30,0.8)
2   [1] 21 22 23 24 24 25 25 26 27 30
```

Rozkład hipergeometryczny

Cztery funkcje do pracy z rozkładem hipergeometrycznym w R:

- `rhyper(N, m, n, k)` Ogólnie rzecz biorąc, odnosi się do funkcji generowania liczb losowych przy danych parametrach i wielkości próby,
  item `phyper(x, m, n, k)` definiuje funkcję rozkładu hipergeometrycznego,
- `dhyper(x, m, n, k)` definiuje funkcję prawdopodobieństwa rozkładu hipergeometrycznego,
- `qhyper(N, m, n, k)` jest funkcją kwantyla rozkładu hipergeometrycznego, która służy do wyznaczania ciągu prawdopodobieństw od 0 do 1.

Tutaj, `x` reprezentuje zbiór wartości, `m` wielkość populacji, `n` liczbę próbek, `k` liczbę elementów w populacji, a `N` wartości o rozkładzie hipergeometrycznym.

Rozkład hipergeometryczny – przykład 1

Príklad.
Pięcioosobowa komisja zostanie wybrana z grupy 10 kobiet i 8 mężczyzn. Jakie jest prawdopodobieństwo, że w skład komisji wchodzą 3 kobiety i 2 mężczyzn? Jakie jest prawdopodobieństwo, że większość członków komisji stanowią kobiety?

Rozkład hipergeometryczny – rozwiązanie

Zgodnie z wymaganiami, $x = 3$ jest liczbą kobiet w komitecie, $m = 10$ jest całkowitą liczbą kobiet w grupie, $n = 8$ jest całkowitą liczbą mężczyzn w grupie, a $k = 5$ jest liczbą członków komitetu.

W związku z tym mamy

```
1 > dhyper(3,10,8,5)
2 [1] 0.3921569
```

Rozkład hipergeometryczny – rozwiązanie

Kobiety mogą mieć większość w komisji, jeśli jest w niej 5, 4 lub 3 kobiety, lub jeśli jest w niej nie więcej niż 2 mężczyzn.

Możemy użyć sumy wartości funkcji `dhyper()`:

```
> dhyper(5,10,8,5)+dhyper(4,10,8,5)+dhyper(3,10,8,5)
[1] 0.6176471
```

Alternatywnie, możemy obliczyć to prawdopodobieństwo za pomocą funkcji `phyper()`, gdzie $x = 2$ mężczyźni w panelu, $m = 8$ całkowita liczba mężczyzn w panelu, $n = 10$ całkowita liczba kobiet w panelu, oraz $k = 5$ liczba członków panelu.

```
> phyper(2,8,10,5)
[1] 0.6176471
```

Rozkład hipergeometryczny – przykład 2

Príklad.
Załóżmy, że w dostawie 100 odtwarzaczy DVD znajduje się dziesięć wadliwych odtwarzaczy. Inspektor wybiera losowo 15 jednostek do kontroli. Zasymulujmy, ilu wadliwych graczy zostanie wybranych w sekwencji 10 kontroli.

Rozkład hipergeometryczny – rozwiązanie

Przesyłka zawiera $m = 10$ wadliwych odtwarzaczy DVD i $n = 90$ wadliwych odtwarzaczy DVD, a inspektor losowo wybiera $k = 15$, kontrola jest powtarzana $N = 10$ razy.

Używamy funkcji `rhyper()` do symulacji ich wyników.

Otrzymujemy więc:

```
1  > rhyper(10,10,90,15)
2   [1] 4 1 1 0 2 0 1 2 3 2
```

Rozkład Pascala

Cztery funkcje do pracy z ujemnym rozkładem Pascalowym w R:

- `rnbinom(N,n,prob)`, gdzie Ž00+ to liczba prób, `N` to wielkość próby, `prob` to prawdopodobieństwo sukcesu. Funkcja ta generuje N zmiennych losowych z zadanym prawdopodobieństwem.
- `pnbinom(x, n, p)` służy do obliczania wartości dysrybuanty rozkładu Pascala. Tutaj `x` jest liczbą niepowodzeń przed `n`tym sukcesem, a `p` jest prawdopodobieństwem sukcesu.
- `dnbinom(x, n, p)` jest prawdopodobieństwem niepowodzenia `x` przed `n`-tym sukcesem (zauważ różnicę), gdy prawdopodobieństwo sukcesu jest równe `p`.
- `qnbinom(x, n, p)` służy do obliczania wartości funkcji kwantyla rozkładu pascala. Tutaj `x` jest wektorem wymaganych poziomów kwantyli, `n` jest całkowitą liczbą prób, a `p` jest prawdopodobieństwem sukcesu na próbę.

Rozkład Pascala – przykłady

Príklad.
Koncern naftowy przeprowadza badania geologiczne, z których wynika, że szanse na znalezienie ropy w odwiercie poszukiwawczym wynoszą 20%. Jakie jest prawdopodobieństwo, że pierwsze odkrycie będzie miało miejsce w trzecim odwiercie? Jakie jest prawdopodobieństwo, że trzeci odwiert zakończony sukcesem będzie siódmym wykonanym odwiertem?

Rozkład Pascala – rozwiązanie

Musimy wyznaczyć prawdopodobieństwo $\mathbb{P}(X = 2)$ przy $n = 14$.

Zauważ, że technicznie rzecz biorąc jest to geometryczna zmienna losowa, ponieważ szukamy tylko jednego sukcesu.

Biorąc pod uwagę implementację funkcji `dnbinom()`, określamy `x=2` niepowodzenie przed `n=1` sukcesem i `p=0.2`.

Mamy więc

```
1  > dnbinom(2,1,0.2)
2  [1] 0.128
```

W przypadku drugiego pytania, wybraliśmy `x=4` nieudanych prób zamiast `n=3` udanych prób.

```
1  > dnbinom(4,3,0.2)
2  [1] 0.049152
```

Rozkład Poissona

Cztery funkcje do pracy z rozkładem Poissona w R:

- `dpois(x,l)` oblicza wartość funkcji prawdopodobieństwa $\mathbb{P}(X = x)$ rozkładu Poissona z parametrem λ wprowadzonym jako argument `l`.
- `ppois(x,l)` oblicza dystrybuanty zmiennej losowej o rozkładzie Poissona. Wyznacza prawdopodobieństwo $\mathbb{P}(X \leq x)$, argument `l` jest parametrem rozkładu. Biorąc pod uwagę inny argument `lower.tail=FALSE`, otrzymujemy prawdopodobieństwo $\mathbb{P}(X > x)$.
- `rpois(k,l)` jest używany do generowania liczb losowych z danego rozkładu Poissona, `k` jest liczbą potrzebnych liczb losowych, a `l` jest parametrem rozkładu.
- `qpois(q,l)` służy do generowania kwantyli danego rozkładu Poissona, `q` jest wektorem potrzebnych poziomów kwantyli, a `l` jest parametrem rozkładu.

Rozkład Poissona – przykłady

Príklad.
Pewna rzeka wylewa średnio raz na 100 lat. Oblicz prawdopodobieństwo $k = 0, 1, 2, 3, 4, 5,$ lub 6 w przedziale 100 lat.

Rozkład Poissona – rozwiązanie

Powódź zdarza się raz na 100 lat, więc można ją uznać za zdarzenie rzadkie, a liczba powodzi odpowiada rozkładowi Poissona.

Używamy funkcji `ppois()` dla `x` , który jest wektorem liczb całkowitych od 0 do 6, oraz parametru `l` równego 1 powodzi co 100 lat.

Mamy

```
1  > x<-seq(0:6)
2  > dpois(x,1)
3  [1] 3.678794e-01 1.839397e-01 6.131324e-02 1.532831e-02
4  + 3.065662e-03
5  [6] 5.109437e-04 7.299195e-05
```

Rozkład Poissona – przykład 2

Príklad.
Sprzedawca ubezpieczeń na życie sprzedaje średnio 3 polisy na życie tygodniowo. Oblicz prawdopodobieństwo, że w danym tygodniu sprzeda kilka polis.

Rozkład Poissona – rozwiązanie

"Bezpieczniki wielokrotne" oznaczają "1 lub więcej" bezpieczników.

Musimy obliczyć prawdopodobieństwo $\mathbb{P}(X > 0) = 1 - \mathbb{P}(X \leq 0)$.

Parametr rozkładu wynosi `l`=3.

Używamy funkcji `ppois()` z opcjonalnym argumentem `lower.tail` ustawionym na `FALSE`.

```
1 > ppois(0,3,lower.tail=FALSE)
2 [1] 0.9502129
```

Alternatywnie, możemy użyć `dpois()`:

```
1 > 1-dpois(0,3)
2 [1] 0.9502129
```

Rozkład Poissona – przykład 3

Príklad.
Pewna firma produkuje 300 silników elektrycznych dziennie. Prawdopodobieństwo, że silnik elektryczny jest uszkodzony wynosi 0,01. Zasymulujmy liczbę wadliwych silników produkowanych każdego dnia w ciągu tygodnia roboczego.

Rozkład Poissona – rozwiązanie

Średnia liczba błędów w dziennej produkcji 300 silników wynosi $\lambda = 0,01 \times 300 = 3$.

Do wygenerowania dziennej liczby błędów wykorzystujemy funkcję `rpois()` z argumentami `k`=5 dni roboczych oraz `l`=3.

Otrzymujemy więc

```
1 > rpois(5,3)
2 [1] 3 3 4 2 2
```

Rozkład Poissona – przykład 4

Príklad.
Rozważmy system komputerowy z Poissonowskim strumieniem przychodzących zadań. ze średnią prędkością 2 zapytań na minutę. Jaka jest maksymalna liczba zadań, które powinny dotrzeć w ciągu jednej minuty przy niezawodności 90%?

Rozkład Poissona – rozwiązanie

Znalezienie maksimum przylotów z co najmniej 90-procentową pewnością oznacza znalezienie kwantyla 90-procentowego.

Używamy funkcji `qpois()` z argumentami `q=0.9` i `l=2` średnia liczba żądań na minutę.

Otrzymujemy więc

```
1 > qpois(0,9,2)
2 [1] 4
```

Rozkłady ciągłe

Wymieńmy niektóre z nich:

- rozkład jednostajny,
- rozkład wykładniczy,
- rozkład normalny,
- rozkład Studenta t,
- rozkład Chi-kwadrat,
- rozkład Fishera F.

Wiele innych dystrybucji jest zaimplementowanych w R.

Rozkłady ciągłe

Wymieńmy niektóre z nich:

- rozkład jednostajny,
- rozkład wykładniczy,
- rozkład normalny,
- rozkład Studenta t,
- rozkład Chi-kwadrat,
- rozkład Fishera F.

Wiele innych dystrybucji jest zaimplementowanych w R.

Zajmiemy się, trzema "błękitnymi" dystrybucjami.

Rozkład jednostajny

Cztery funkcje do pracy z rozkładem jednostajnym w R:

- `dunif()`, który definiuje funkcję gęstości, której argumentami są wektor `x` oraz parametry `min` i `max` rozkładu,
- `punif()`, która definiuje dystrybuantu rozkładu, jej argumentami są wektor `x` oraz parametry `min` i `max` rozkładu,
- `qunif()`, która dostarcza funkcję kwantyla, której argumentami są kwantyle `q` oraz parametry `min` i `max` rozkładu,
- `runif()`, który generuje losowe wartości zmiennej, jego argumentami są wielkość próby `n` oraz parametry `min` i `max` rozkładu.

Rozkład jednostajny - przykład

Príklad.
Załóżmy, że tramwaje odjeżdżają z dworca w regularnych pięciominutowych odstępach. Oblicz prawdopodobieństwo, że pasażer będzie czekał:

a) więcej niż 3 minuty,
b) nie dłużej niż 1,5 minuty,

jeśli przybędzie na przystanek w przypadkowym momencie.

Rozkład jednostajny – rozwiązanie a)

Czas oczekiwania jest zmienną losową, która ma rozkład jednostajny o parametrach $a = 0$ i $b = 5$.

Zatem prawdopodobieństwo, że pasażer będzie czekał dłużej niż 3 minuty wynosi $\mathbb{P}(X > 3) = 1 - F(3)$.

Otrzymujemy

```
1 > 1-punif(3,min=0,max=5)
2 [1] 0.4
```

Rozkład jednostajny – rozwiązanie b)

Pytanie (b) dotyczy prawdopodobieństwa $\mathbb{P}(X \leq 1,5) = F(1,5)$.

Otrzymujemy pożądany wynik jako `puinf(1,5,min=0,max=5)`, więc prawdopodobieństwo wynosi $0,3$.

```
1 > punif(1.5,min=0,max=5)
2 [1] 0.3
```

Rozkład jednostajny – symulacja

Możemy zasymulować tę sytuację za pomocą funkcji `runif()`.

Poprzez zwiększenie liczebności próby możemy również zilustrować, jak zwiększenie liczby losowych eksperymentów prowadzi do lepszego przybliżenia rozkładu.

Aby wyświetlić wykres, należy uruchomić kod

```
1 par(mfrow = c(3, 1))
2 hist(runif(10,min=0,max=5))
3 hist(runif(100,min=0,max=5))
4 hist(runif(1000,min=0,max=5))
```

Rozkład wykładniczy

Cztery funkcje do pracy z rozkładem wykładniczym w R:

- `dexp()`, która jest funkcją gęstości, której argumentami są wektor `x` i parametr `rate` rozkładu,
- `pexp()`, która jest dystrybuanta rozkładu, której argumentami są wektor `x` oraz parametr `rate` rozkładu,
- `qexp()`, określający funkcję kwantylową, której argumentami są kwantyle `q` oraz parametr `rate` rozkładu,
- `rexp()`, który generuje losowe wartości zmiennej, jego argumentami są wielkość próby `n` oraz parametr `rate` rozkładu.

Rozkład wykładniczy - przykład

Príklad.
Załóżmy, że średni czas obsługi przy kasie w supermarkecie wynosi trzy minuty. Znaleźć prawdopodobieństwo, że kasjerka skończy obsługiwać klienta w czasie:

a) mniej niż dwie minuty,
b) więcej niż pięć minut.

Rozkład wykładniczy – rozwiązanie

Średni czas obsługi w kasie jest równy odwrotności częstotliwości obsługi,

Zatem częstotliwość obsługi klientów wynosi $\frac{1}{3}$ klientów na minutę. Odpowiedzią na pytanie (a) jest więc prawdopodobieństwo $\mathbb{P}(X < 2)$.

```
1 > pexp(1/3,2)
2 [1] 0.4865829
```

Odpowiedzią na pytanie (b) jest prawdopodobieństwo $\mathbb{P}(X > 5)$.

```
1 > pexp(1/3,2)
2 [1] 0.4865829
```

Rozkład wykładniczy – przykład 2

Príklad.
Wiadomo, że awarie pewnego typu sprzętu elektronicznego mają rozkład wykładniczy ze średnim czasem 30 miesięcy zanim sprzęt ulegnie awarii. Znajdźmy prawdopodobieństwo, że.

a) losowo wybrane urządzenie ulega awarii w ciągu pierwszego roku (12 miesięcy),
b losowo wybrane urządzenie działa dłużej niż 6 lat (72 miesiące).

Rozkład wykładniczy – rozwiązanie a)

Zmienna losowa, która reprezentuje czas do awarii urządzenia jest oznaczana przez X.

Musimy odpowiedzieć na pytanie, jakie jest prawdopodobieństwo $\mathbb{P}(X < 12)$, jeśli zmienna losowa X ma rozkład wykładniczy z parametrem $\lambda = 1/30$.

Wynik uzyskuje się za pomocą polecenia:

```
1 > pexp(1/30,12)
2 [1] 0.32968
```

Rozkład wykładniczy – rozwiązanie b)

Aby odpowiedzieć na pytanie b), musimy znaleźć prawdopodobieństwo $\mathbb{P}(X \geq 70)$.

Aby uzyskać odpowiedź za pomocą funkcji `pexp()`, musimy ustawić argument `lower.tail=FALSE`.

Otrzymujemy

```
1 > pexp(1/30,72,lower.tail=FALSE)
2 [1] 0.09071795
```

Rozkład wykładniczy – kwantyle

Aby zilustrować znaczenie kwantyli, określamy czas, po jakim 60% urządzeń przestaje działać.

Używamy funkcji `qexp()` jak pokazano w poniższym poleceniu:

```
1  > qexp(0,6,1/30)
2  [1] 27.48872
```

60% sprzętu ulegnie awarii w ciągu około 27,5 miesiąca.

rozkład normalny

Cztery funkcje do pracy z rozkładem normalnym w R:

- `dnormf()`, która jest funkcją gęstości, której argumentami są wektor `x` oraz parametry `mean` i `sd` rozkładu,
- `pnorm()`, który jest dystrybuanta rozkładu, której argumentami są wektor `x` oraz parametry `mean` i `sd` rozkładu
- `qnorm()`, który reprezentuje funkcję kwantylową, której argumentami są kwantyle `q` oraz parametry `mean` i `sd` rozkładu,
- `rnorm()`, który generuje wartości losowe zmiennej, jego argumentami są wielkość próby `n` oraz parametry `mean` i `sd` rozkładu.

Rozkład normalny - przykład

Príklad.
Załóżmy, że wyniki egzaminu wstępnego na studia mają rozkład normalny. Średni wynik tego testu wynosi 70, a odchylenie standardowe 10. Jaki jest procent uczniów

a) , którzy uzyskają na egzaminie wynik co najmniej 85 punktów,
b) , którzy uzyskali na egzaminie nie więcej niż 60 punktów.

Rozkład normalny – rozwiązanie a)

Używamy funkcji \pnorm() rozkładu normalnego o średniej 70 i odchyleniu standardowym 10. Interesuje nas $\mathbb{P}(X \geq 85)$, górna granica rozkładu normalnego. Dlatego używamy parametru logicznego `lower.tail=FALSE`.

Mamy

```
1 > pnorm(85, mean=70, sd=10, lower.tail=FALSE)
2 [1] 0.0668072
```

Rozkład normalny – rozwiązanie b)

Aby odpowiedzieć na pytanie (b), musimy obliczyć prawdopodobieństwo $\mathbb{P}(X < 60)$.

Ponownie korzystamy z funkcji `pnorm()`:

```
1 > pnorm(60, mean=70, sd=10)
2 [1] 0.1586553
```

Rozkład normalny - przykład 2

Príklad.
Według danych z www.uvzsr.sk, średni wzrost 18-letnich chłopców na Słowacji w 2011 roku wynosił 179 cm z odchyleniem standardowym 6,68 cm. Przyjmij, że wzrost ma rozkład normalny i określ prawdopodobieństwo, że losowo wybrany chłopiec w wieku 18 lat jest

a) jest wyższy niż 200 ,cm,
b) mniej niż 160 cm.

Rozkład normalny – rozwiązanie

Oznaczmy zmienną losową, która opisuje wysokość jako X,

Aby odpowiedzieć na pytanie (a), musimy obliczyć prawdopodobieństwo $\mathbb{P}(X \geq 200)$.

W (b) musimy znaleźć $\mathbb{P}(X < 160)$.

Korzystając z funkcji `pnorm()` otrzymujemy wartość

```
1 > pnorm(200,179,6.68,lower.tail=FALSE)
2 [1] 0.000834096
3 > pnorm(160,179,6.68)
4 [1] 0.002225376
```

Statystyka i programowanie w R

IV. Programowanie w R

Funkcje

Prawie wszystkie działania w R są wykonywane za pomocą funkcji.

Zaimplementowany jest bogaty zestaw funkcji wbudowanych.

Użytkownik może zdefiniować dodatkowe funkcje

Wbudowane funkcje można podzielić na

- Funkcje matematyczne,
- Funkcje łańcuchów elementów,
- Wyspecjalizowane funkcje statystyczne i probabilistyczne,
- Inne przydatne funkcje.

Funkcje matematyczne

O niektórych z nich wspomnieliśmy już w lekcji 1.

Tutaj podamy kilka dodatkowych szczegółów

Funkcja logarytmiczna `log()` oblicza logarytm naturalny jako wartość domyślną.

Aby otrzymać logarytm o dowolnej podstawie, należy zadeklarować argument `base` funkcji `log()`.

```
1 > log(4)
2 [1] 1.386294
3 > log(4,base=2)
4 [1] 2
```

Funkcje matematyczne

Funkcje trygonometryczne działają z argumentem podanym w radianach.

Gdy używamy stopni, musimy przekształcić wartość jako $r = \frac{\pi \cdot \alpha}{180}$, gdzie r jest nową miarą w radianach, a α jest starą wartością w stopniach, lub możemy również użyć funkcji `deg2rad()` z pakietu `REdaS`.

```
> library(REdaS)
> sin(90)
[1] 0.8939967
> sin(deg2rad(90))
[1] 1
```

```
> tan(45)
[1] 1.619775
> tan(deg2rad(45))
[1] 1
```

Funkcje matematyczne – liczby zespolone

Funkcje do obliczeń z liczbami złożonymi

- `Re(z)` Część rzeczywista z.
- `Im(z)` Część urojona z.
- `Mod(z)` Moduł z.
- `Arg(z)` Argument z.
- `Conj(z)` liczba zespolona *overlinez*.

Funkcje tekstowe

Funkcja nchar()+ określa rozmiar każdego elementu wektora znaków.

```
1  > z<-c("yellow","black","white")
2  > nchar(z)
3  [1] 6 5 5
4  > str<-"This␣is␣a␣long␣string"
5  > nchar(str)
6  [1] 21
```

Funkcje teksowe

Argument keepNA jest wartością logiczną, która określa, czy zwracać NA, gdy NA jest zawarte w x.

```
1 > z<-c("",NULL, "black",NA)
2 > nchar(z,keepNA=TRUE)
3 [1] 0 5 NA
4 > nchar(z,keepNA=FALSE)
5 [1] 0 5 2
```

Funkcje tekstowe

Lista funkcji tekstowych

- `nchar()` Liczba znaków w łańcuchu.
- `substr()` Wybierz lub zamień podłańcuchy.
- `grep()` Znajdowanie wzorca w łańcuchu znaków.
- `strsplit()` Tworzy łańcuch w podanym punkcie podziału.
- `sub()` Znajduje wzorzec w tekste i zastępuje go.
- `paste()` Łączy teksty używając podanego separatora.
- `toupper()` Konwertuje ciąg znaków na wielkie litery.
- `tolower()` Konwertuje tekst na małe litery.

Funkcje teksowe

Aby znaleźć określony wzór w tekste, użyj `grep()`.

```
> str <- c('abcd','bdcd','abcdabcd')
> pattern<- 'abc'
> grep(pattern, str)
[1] 1 3
> pattern<- 'Abc'
> grep(pattern, str)
integer(0)
> grep(pattern, str,ignore.case=TRUE)
[1] 1 3
> pattern<- 'a*'
> grep(pattern, str)
[1] 1 2 3
> grep(pattern, str,fixed=TRUE)
integer(0)
```

Funkcje tekstowe

Aby zastąpić znaleziony wzorzec innym tekstem, użyj funkcji `sub()`.

```
1 > str<-"Bohemia␣does␣not␣use␣EURO␣currency"
2 > str<-sub("Bohemia","Czechia",str)
3 > str
4 [1] "Czechia␣does␣not␣use␣EURO␣currency"
```

Istnieje opcjonalny argument `ignore.case`,

Funkcje tekstowe

Inną funkcją służącą do manipulowania łańcuchem tekstowym jest `substr()`.

Przyjmuje trzy argumenty: łańcuch tekstowy `x` oraz `start` i `stop`, aby zadeklarować pozycję pierwszego i ostatniego znaku, który ma zostać wybrany lub zastąpiony.

```
1 > str<-"Bohemia␣does␣not␣use␣EURO␣currency"
2 > substr(str,1,7)
3 [1] "Bohemia"
4 > substr(str, 1, 5)<-"Czech"
5 > str
6 [1] "Czechia␣does␣not␣use␣EURO␣currency"
```

Funkcje tekstowe

Funkcja `strsplit()` rozdziela elementy wektora znaków `x` na pozycje określone przez drugi argument `split`.

```
> strsplit(str,"")
[[1]]
[1]"C" "z" "e" "c" "h" "i" "a" "␣" "d" "o" "e" "s" "␣" "n" "o" "t" "␣" "u"
[20] "e" "␣" "E" "U" "R" "O" "␣" "c" "u" "r" "r" "e" "n" "c" "y"
> strsplit(str, "␣")
[[1]]
[1] "Czechia" "does" "not" "use" "EURO" "currency"
> strsplit(str, "e")
[[1]]
[1] "Cz" "chia␣do" "s␣not␣us" "␣EURO␣curr" "ncy"
```

Funkcje tekstowe

Funkcja `strsplit()` rozdziela elementy wektora znaków `x` na pozycje określone przez drugi argument `split`.

Wspomnieliśmy już o funkcji `paste()` służącej do łączenia ciągów znaków.

Argumentami są łańcuchy do konkatenacji oraz separator, który definiuje ich separator.

```
1 > paste("x",1:4,sep="")
2 [1] "x1" "x2" "x3" "x4"
3 > paste("Today␣is",date(),sep="␣")
4 [1] "Today␣is␣Tue␣Apr␣27␣10:39:55␣2021"
5 > paste(c("a","b"),1:4,sep="/")
6 [1] "a/1" "b/2" "a/3" "b/4"
```

Funkcje tekstowe

Dwie powiązane funkcje, `toupper()` i `tolower()`, przekształcają podany tekst na duże i małe litery.

```
1 > toupper(str)
2 [1] "CZECHIA␣DOES␣NOT␣USE␣EURO␣CURRENCY"
3 > tolower(str)
4 [1] "czechia␣does␣not␣use␣euro␣currency"
```

Elementarne funkcje statystyczne

- `mean()` Średnia próbna.
- `median()` Mediana próby.
- `sd()` Odchylenie standardowe.
- `var()` Wariancja próby .
- `mad()` Odchylenie bezwzględne mediany.
- `quantile()` Kwantyle próbky, kwartyle są domyślne.
- `range()` Zakres wartości.
- `sum()` Suma elementów wektora.
- `min()` Minimum.
- `max()` Maximum.

Elementarne funkcje statystyczne – `mean()` argumenty opcjonalne

`trim`, który określa procent najwyższych i najniższych wartości, które są pomijane w obliczeniach, zwracając w ten sposób przyciętą średnią.

Drugi opcjonalny argument `na.rm` jest wartością logiczną, która określa, czy usunąć wartości `NA` przed kontynuowaniem obliczeń.

```
1  > x<-c(1,3,5,10,12)
2  > mean(x)
3  [1] 6.2
4  > mean(x,trim=0,2)
5  [1] 6
```

```
1  > x<-c(1,5,2,12,NA,3,6)
2  > mean(x)
3  [1] NA
4  > mean(x,na.rm=TRUE)
5  [1] 4.833333
6  > mean(x,na.rm=TRUE,trim=0.17)
7  [1] 4
```

Elementarne funkcje statystyczne – quantiles()

Domyślnym wynikiem są kwartyle

Aby określić poziomy prawdopodobieństwa dla kwantyli, opcjonalny argument prob musi być podany jako wektor liczbowy.

```
1  > delay<-c(0,9,0,42,14,0,11)
2  > quantile(delay)
3    0% 25% 50% 75% 100%
4   0.0 0.0 9.0 12.5 42.0
5   > quantile(delay,prob=c(0,0,33,0,67,1))
6     0% 33% 67% 100%
7   0.00 0.00 11.06 42.00
```

Elementarne funkcje statystyczne – `mad()`

Bezwzględne odchylenie mediany jest solidną miarą zmienności jednoczynnikowej próbki danych ilościowych.

Dla próbki $X_1, \ldots, X_n$ jest ona określona wzorem:

$$\mathrm{MAD}(X) = \mathrm{median}\{|X_i - \overline{X}|\}$$

```
1  > mad(delay)
2  [1] 13.3434
```

Użyteczne funkcje – seq()

Funkcja `seq()` generuje ciąg liczb zaczynający się od `from` i kończący się `to`. Ostatni argument `by` określa krok sekwencji.

```
> seq(10)
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(5,15)
 [1]  5  6  7  8  9 10 11 12 13 14 15
> seq(5,15,2)
[1]  5  7  9 11 13 15
```

Użyteczne funkcje – rep()

Funkcja rep() przyjmuje dwa argumenty, wektor x do powtórzenia oraz liczbę cykli powtórzeń n.

```
> rep(1,10)
 [1] 1 1 1 1 1 1 1 1 1 1
> rep(c(1,3),4)
[1] 1 3 1 3 1 3 1 3
> rep("hello",3)
[1] "hello" "hello" "hello"
```

Użyteczne funkcje – `sort()` i `order()`

.

Funkcje `sort()` i `order()` związane są z porządkowaniem elementów wektora `x`.

`sort()` dostarcza posortowanych rosnąco wartości, podczas gdy `order()` dostarcza indeksów posortowanych komponentów w oryginalnym wektorze.

```
1  > x<-c(5,2,10,3,7,8)
2  > sort(x)
3  [1]  2  3  5  7  8 10
4  > order(x)
5  [1]  2  4  1  5  6  3
```

Użyteczne funkcje – `rev()`

Daje wektor `x` w odwrotnej kolejności elementów

```
> rev(x)
[1]  8  7  3 10  2  5
> rev(sort(x))
[1] 10  8  7  5  3  2
```

Komendy warunkowe – `if`

`if()` wykonuje operacje na podstawie prostego warunku

```
if (warunek) {polecenie do wykonania, jeśli warunek jest spełniony}
```

Więcej niż jedno stwierdzenie musi być w nawiasie

```
1  > x<-5
2  > if(x%%2){print("Odd␣number")}
3  [1] "Odd␣number"
4  > x<-6
5  > if(x%%2){print("Odd␣number")}
6  >
```

Komendy warunkowe – `if ... else`

To rozszerzenie komendy `if` ma ogólną składnię w postaci:

```
if (test_expression) {
příkaz1
} else {
příkaz2
}
```

```
1 > x<-5
2 > if(x%%2){print("Odd number")} else {print ("Even number")}
3 [1] "Odd number"
4 > x<-10
5 > if(x%%2){print("Odd number")} else {print ("Even number")}
6 [1] "Even number"
```

Komendy warunkowe – `if ... else`

Możemy dalej dostosowywać poziom kontroli poprzez zagnieżdżanie instrukcji else if. Dzięki else if możemy dodać dowolną ilość warunków. Składnia jest następująca:

```
if (warunek1) {
    polecenie1
    } else if (warunek2) {
    komenda2
    } else if (warunek3) {
    komenda3
    } else {
    komenda4
}
```

Komendy warunkowe – przykład `if... else`

Príklad.
Stawki podatku VAT różnią się w zależności od zakupionego produktu. Załóżmy, że mamy trzy różne rodzaje produktów z różnymi stawkami VAT (które faktycznie obowiązują na Słowacji):

Kategoria	**towary**	**VAT**
A	Maseczki, respiratory (w rzeczywistości zwolnione z VAT)	0%.
B	Wybrana żywność, książki, czasopisma, leki.	10%
C	Wszystkie pozostałe	20%

Napisz polecenie, które zastosuje właściwą stawkę VAT do produktu, który został zakupiony przez klienta.

Komendy warunkowe – `if ... else` rozwiązanie

```
> category <- "B"
> price<-50
> if (category =="A"){
  cat("A vat rate of 0% is applied.","The total price is",price *1.00)
} else if (category =="B"){
    cat("A vat rate of 10% is applied.","The total price is",price *1.10)
} else {
    cat("A vat rate of 20% is applied.","The total price is",price *1.20)
}
A vat rate of 10% is applied. The total price is 55
```

Komendy warunkowe – `ifelse`

Komendy `if` i `if ... else` nie powinny być używane, jeżeli warunek jest obliczany na wektor.

Polecenie `if` ocenia warunek tylko dla pierwszego elementu wektora.

```
1  > x<-c(5,4,3,2,1)
2  > if(x>3){x*2}
```

możemy się spodziewać, że wynik będzie następujący: `10,8,3,2,1`. Jednakże rzeczywisty wynik jest następujący:

```
1  [1] 10  8  6  4  2
2  Warning message:
3  In if (x > 3) { :
4    the condition has length > 1 and only the first element
5     will be used
```

Komendy warunkowe – `ifelse`

Aby uzyskać oczekiwany rezultat, musimy użyć polecenia `ifelse` o ogólnej składni:

```
ifelse(warunek, wyrażenie1, wyrażenie2)
```

```
1  > ifelse(x>3,2*x,x)
2  [1] 10  8  3  2  1
```

Komendy warunkowe – switch

`switch()` testuje wyrażenie względem elementów listy. Każda wartość na liście jest nazywana case.

Składnia funkcji `switch()`:

```
switch (wyrażenie, lista)
```

```
1  > x<-10
2  > switch(x%%2+1, "even", "odd")
3  [1] "even"
4  > x<-9
5  > switch(x%%2+1, "even", "odd")
6  [1] "odd"
```

Komendy warunkowe – switch

Jeśli wyrażenie jest łańcuchem znaków, switch() zwraca wartość opartą na nazwie elementu.

```
> x <- "a"
> switch(x, "a"="apple", "b"="banana", "c"="cherry")
[1] "apple"
> x <- "c"
> switch(x, "a"="apple", "b"="banana", "c"="cherry")
[1] "cherry"
```

Komendy warunkowe – `switch`

W przypadku wielokrotnego dopasowania, zwracana jest wartość pierwszego pasującego elementu.

Możemy zdefiniować wartość domyślną, która zostanie zwrócona, jeśli nie zostanie znalezione dopasowanie.

```
> x <- "a"
> switch(x, "a"="apple", "a"="apricot", "a"="avocado")
[1] "apple"
> x <- "x"
> switch(x, "a"="apple", "b"="banana", "c"="cherry","some␣fruit")
[1] "some␣fruit"
```

Pętla – `for`

Pętla \`for` pozwala nam na ustaloną liczbę powtórzeń danej instrukcji lub bloku instrukcji.

Ogólna składnia pętli `for` jest następująca:

```
for (val in sequence)
{
polecenie
}
```

gdzie `sequence` jest wektorem, a `val` przyjmuje każdą z wartości sekwencji podczas pętli.

Pętla – for

```
1  > x<-c(2,5,10,8,6,3,12)
2  > limit<-mean(x)
3  > count<-0
4  > for(i v x){
5  if (i>limit) count<-count+1
6    }
7  > count
8  [1] 3
```

Pętla – `for`

Możemy zatrzymać pętlę zanim przejdzie ona przez wszystkie elementy używając `break`.

```
> x<-c(2,4,6,5,8,10,11,12,14,20)
> for (i in x){
    if(i%%2==1) {break}
  print(i/2)
  }
[1] 1
[1] 2
[1] 3
```

Pętla – `for`

Używając `next` możemy pominąć iterację bez kończenia pętli.

```
> x<-c(2,4,5,8,11,20)
> for (i in x) {
+ if(i%%2==1) {next}
+ print(i/2)
+ }
[1] 1
[1] 2
[1] 4
[1] 10
```

Pętla – `while`

Przydatne, gdy chcemy powtarzać komendę lub blok komend do momentu spełnienia pewnego warunku.

```
while (condition){
 polecenia
}
```

Pętla – `while`

Użyj pętli `while` do symulowania rzutu kością aż do uzyskania pierwszych sześciu punktów.

```
> roll<-0
> while(roll!=6){
  roll<-sample(1:6,1)
  print(roll)
  }
[1] 1
[1] 4
[1] 3
[1] 4
[1] 6
```

Pętla – repeat

Podobna do pętli `while`, ale blok instrukcji jest wykonywany przynajmniej raz, niezależnie od tego, czy warunek jest spełniony.

```
repeat{
polecenie
}
```

Pętla repeat nie sprawdza warunku zakończenia pętli. Warunek musi być jawnie wstawiony do ciała pętli, a instrukcja break musi być użyta do zakończenia pętli.

Pętla – repeat

Użyj pętli repeat do symulowania rzutu kością aż do pierwszego rzutu o wartości sześciu punktów.

```
> repeat{
  roll<-sample(1:6,1)
  print(roll)
  if(roll==6){break}
  }
[1] 5
[1] 2
[1] 1
[1] 5
[1] 6
```

Funkcja zdefiniowana przez użytkownika

Ogólna struktura funkcji jest następująca

```
myfunction_name <- function(arg1, arg2, ... ){
polecenia
return(obiekt)
}
```

Funkcja zdefiniowana przez użytkownika

Poszczególnymi składnikami funkcji są:

- **Nazwa funkcji**, który jest rzeczywistą nazwą funkcji. Jest on przechowywany w środowisku R jako obiekt o tej nazwie.
- **Argumenty**, które są symbolami wieloznacznymi. Kiedy wywołujemy funkcję, przekazujemy wartości argumentów. Argumenty są opcjonalne, to znaczy, że funkcja nie musi zawierać żadnych argumentów. Argumenty mogą mieć również wartości domyślne.
- **Ciało funkcji**, który zawiera zestaw poleceń określających działanie funkcji. Ciało funkcji znajduje się wewnątrz nawiasów złożonych.
- **Wartość zwracana**, który jest ostatnim obliczanym wyrażeniem w treści funkcji.

Funkcja zdefiniowana przez użytkownika

Zdefiniujmy funkcję `cubes()`, która wypisuje trzecią potęgę liczb w ciągu.

```
1  cubes <- function(a) {
2     for(i in 1:a) {
3        b <- i^3
4        print(b)
5     }
6  }
```

```
1  > cubes(6)
2  [1] 1
3  [1] 8
4  [1] 27
5  [1] 64
6  [1] 125
7  [1] 216
```

Funkcja zdefiniowana przez użytkownika

Funkcja może być zdefiniowana bez argumentów. W tych okolicznościach tworzy ciąg trzecich potęg o stałej długości.

```
1  cubes <- function() {
2     for(i v 1:5) {
3        b <- i^3
4        print(b)
5     }
6  }
```

```
1  > cubes()
2  [1] 1
3  [1] 8
4  [1] 27
5  [1] 64
6  [1] 125
```

Funkcja zdefiniowana przez użytkownika

Argumenty wywołania funkcji mogą być wprowadzone w takiej samej kolejności, w jakiej zostały zdefiniowane w funkcji.

```
1  cubes <- function(start,end) {
2     for(i in start:end) {
3        b <- i^3
4        print(b)
5     }
6  }
```

```
1  > cubes(12,10)
2  [1] 1728
3  [1] 1331
4  [1] 1000
```

Funkcja zdefiniowana przez użytkownika

Alternatywnie, możemy wywołać funkcję przez nazwy argumentów

```
1  > cubes(end=12,start=10)
2  [1] 1000
3  [1] 1331
4  [1] 1728
```

Funkcja zdefiniowana przez użytkownika

Funkcja cubes() może być zdefiniowana z domyślnymi argumentami

```
1  cubes <- function(start=1,end=10) {
2     for(i in start:end) {
3        b <- i^3
4        print(b)
5     }
6  }
```

```
1  > cubes(end=4)
2  [1] 1
3  [1] 8
4  [1] 27
5  [1] 64
```

Funkcja zdefiniowana przez użytkownika

Co się stanie, gdy będziemy chcieli wstawić do zmiennej wartość `cubes(2,2)`?

```
1 > z<-cubes(2,2)
2 [1] 8
3 > z
4 NULL
```

Zmienna `z` nie zawiera wartości

Funkcja zdefiniowana przez użytkownika

Funkcja musi być zdefiniowana przy użyciu wartości zwracanej +return()+.

```
1  cubes <- function(start=1,end=10) {
2     for(i in start:end) {
3        b <- i^3
4        return(b)
5     }
6  }
7  > z<-cubes(2,2)
8  > z
9  [1] 8
```

Funkcja zdefiniowana przez użytkownika

Funkcja `cubes()` zwraca tylko jedną wartość

Jeśli chcemy rozszerzyć wynik na cały zakres, musimy zdefiniować zmienną wyjściową jako wektor.

```
1  cubes <- function(start=1,end=10) {
2      b<-vector() # inicjalizacja wektora
3      for(i in start:end) {
4          b[i-start+1]<-i^3 # zmien indeks
5      }
6      return(b)
7  }
```

Funkcja zdefiniowana przez użytkownika

Teraz otrzymujemy pełny ciąg trzecich potęg w podanym zakresie:

```
1 > z<-cubes(4,8)
2 > z
3 [1] 64 125 216 343 512
```

Funkcja zdefiniowana przez użytkownika

W programowaniu R, funkcje nie zwracają wielu wartości.

Możemy jednak stworzyć listę zawierającą wiele obiektów, które funkcja powinna zwrócić.

```
1  powers<-function(start=1,end=10) {
2          b<-wektor()
3          c<-wektor()
4          for(i in start:end) {
5          b[i-start+1]<-i^2
6          c[i-start+1]<-i^3
7      }
8      out<-list(b,c)
9      return(out)
10 }
```

Funkcja zdefiniowana przez użytkownika

Teraz możemy użyć go, aby uzyskać wyjście w postaci listy

```
> powers(1,5)
[[1]]
[1]  1  4  9 16 25

[[2]]
[1]   1   8  27  64 125
```

Uruchamianie skryptów w R

Skrypt R jest po prostu plikiem tekstowym zawierającym (prawie) te same polecenia, które wpisałbyś pisząc

Można go utworzyć w dowolnym prostym edytorze tekstu i zapisać z rozszerzeniem `.R`.

Istnieją zasadniczo dwa polecenia do uruchomienia skryptu w Linuksie.

```
Rscript filename.R
```

co jest preferowane. Starsze polecenie to

```
R CMD BATCH filename.R
```

Statystyka i programowanie w R

V. Podstawy grafiki w R

Wykresy punktowe

Tworzymy je po prostu za pomocą funkcji `plot()`.

W najprostszym ujęciu funkcja przyjmuje dwa argumenty `x` oraz `y`.

Zmienne te są wektorami, które zawierają wartości, które chcemy wykreślić.

Długość wektorów musi być taka sama.

Wykresy punktowe

Príklad.
Przypuśćmy, że lokalna lodziarnia śledzi, ile lodów sprzedaje w zależności od temperatury w południe danego dnia. Oto ich dane za ostatnie 10 dni:

Temperature	28	30.2	32	31	29.5	26	31.5	30	29	34
Sprzedaż (euro)	540	560	530	570	525	490	530	530	500	580

Wykresy punktowe

Najpierw definiujemy dwa wektory liczbowe:

`x` który zawiera temperatury

`y` który będzie reprezentować dzienną wielkość sprzedaży

Następnie rysujemy wykres punktowy

```
1  > x<-c(28,30.2,32,31,29.5,26,31.5,30,29,34)
2  > y<-c(540,560,530,570,525,490,530,530,500,580)
3  > plot(x,y)
```

Jak zapisać wykres

Za pomocą funkcji `dev.copy()` możemy zapisać zawartość okna graficznego do pliku bez konieczności ponownego wprowadzania komend.

Aby utworzyć plik `newplot.png` z naszego wykresu, wpisujemy:

```
1 > dev.copy(png,'newplot.png')
2 > dev.off()
```

Jak zapisać obraz

Alternatywnie, możemy przekierować wyjście z ekranu do pliku.

Możemy użyć funkcji

`pdf()`	Vector `pdf` format, najlepszy wybór, gdy jest używany z `pdflatex`.
`svg()`	Format wektorowy `svg`; łatwy do zmiany rozmiaru.
`postscript()`	Format wektorowy postscriptowy, łatwa zmiana rozmiaru.
`png()`	Format bitmap o wysokiej rozdzielczości, nie może być zmieniany bez utraty rozmiaru.
`jpeg()`	Skompresowany format bitmapy, bez bezstratnej zmiany rozmiaru.
`bmp()`	Format bitmap o wysokiej rozdzielczości, nie zmienia rozmiaru bezstratnie.
`tiff()`	format bitmap o wysokiej rozdzielczości, nie zmienia rozmiaru bezstratnie.

Opcje przechowywania wykresów

`filename`	Nazwa zapisywanego pliku, w razie potrzeby z pełną ścieżką.
`width`	Szerokość wynikowego wykresu, wartość domyślna 7 in.
`height`	Wysokość wynikowego wykresu, wartość domyślna 7 in.
`res`	rozdzielczość obrazu, ważne dla formatów bitmapowych, domyślnie 72 dpi.
`units`	Jednostki miary.
`bg`	Kolor tła.
`fg`	Kolor pierwszego planu.
`family`	Używana czcionka (domyślnie Helvetica).

Modyfikacja wykresu – punkty znacznikowe

Znacznik punktu jest podawany przez wartość argumentu `pch` funkcji `plot()`.

Możliwe wartości

□	pch=0	○	pch=1	△	pch=2	+	pch=3	×	pch=4
◇	pch=5	▽	pch=6	⊠	pch=7	✳	pch=8	⊕	pch=9
⊕	pch=10	✡	pch=11	⊞	pch=12	⊗	pch=13	◪	pch=14
■	pch=15	●	pch=16	▲	pch=17	◆	pch=18	●	pch=19
•	pch=20	○	pch=21	□	pch=22	◇	pch=23	△	pch=24

Modyfikacja wykresu - etykiety punktów

Spróbujmy zmodyfikować nasz wykres

```
1 > plot(x,y,pch=17)
2 > plot(x,y,pch=1)
```

Modyfikacja wykresu - etykiety punktów

Spróbujmy zmodyfikować nasz wykres

```
1 > plot(x,y,pch=17)
2 > plot(x,y,pch=1)
```

Co edytować dalej?

Modyfikacja wykresu - etykiety punktów

Spróbujmy zmodyfikować nasz wykres

```
1 > plot(x,y,pch=17)
2 > plot(x,y,pch=1)
```

Co edytować dalej? Typ linii łączącej punkty

Modyfikacja wykresu – typ łącza

Typ linii punktowej ustawia się za pomocą argumentu `type` funkcji `plot()`.

Możliwe wartości

`p`	Wykres punktowy, wartość domyślna.
`l`	Linia ciągła.
`b`	Linia ciągła z punktami.
`c`	Części linii ciągłych z pominiętymi punktami.
`o`	Części linii ciągłych, punkty przerysowane.
`h`	Wykres podobny do histogramu.
`s`	Schemat schodów.

Modyfikacja wykresu – typ linii

Spróbujmy zmodyfikować nasz wykres

```
1  > plot(x,y,type="l")
2  > dev.off()
3  > plot(x,y,type="s")
4  > dev.off()
5  > plot(x,y,pch=17,type="b")
6  > dev.off()
```

Modyfikacja wykresu – typ linii

Spróbujmy zmodyfikować nasz wykres

```
1 > plot(x,y,type="l")
2 > dev.off()
3 > plot(x,y,type="s")
4 > dev.off()
5 > plot(x,y,pch=17,type="b")
6 > dev.off()
```

Co edytować dalej?

Modyfikacja wykresu – typ linii

Spróbujmy zmodyfikować nasz wykres

```
1 > plot(x,y,type="l")
2 > dev.off()
3 > plot(x,y,type="s")
4 > dev.off()
5 > plot(x,y,pch=17,type="b")
6 > dev.off()
```

Co edytować dalej? Styl linii łączącej

Modyfikacja wykresu – styl łącza

Styl linii jest ustawiany za pomocą argumentu `lty` funkcji `plot()`.

Możliwe wartości

1	Pogrubiona linia (domyślnie).	2	Pogrubiona linia.
3	Kreska z kropkami.	4	Kreski i przecinki.
5	Długie kreski.	6	D Długie i krótkie podwójne linie przerywane.

Szerokość linii ustawia się za pomocą argumentu `lwd` funkcji `plot()`.

Modyfikacja wykresu – styl łącza

Spróbujmy zmodyfikować nasz wykres

```
1 > plot(x,y,type="l",lty=5)
2 > dev.off()
3 > plot(x,y,type="l",lty=1,lwd=2)
4 > dev.off()
```

Modyfikacja wykresu – styl łącza

Spróbujmy zmodyfikować nasz wykres

```
1 > plot(x,y,type="l",lty=5)
2 > dev.off()
3 > plot(x,y,type="l",lty=1,lwd=2)
4 > dev.off()
```

Co edytować dalej?

Modyfikacja wykresu – styl łącza

Spróbujmy zmodyfikować nasz wykres

```
1 > plot(x,y,type="l",lty=5)
2 > dev.off()
3 > plot(x,y,type="l",lty=1,lwd=2)
4 > dev.off()
```

Co edytować dalej? Kolor

Modyfikacja grafu - kolorowanie

Zanim zaczniemy, jeden problem:

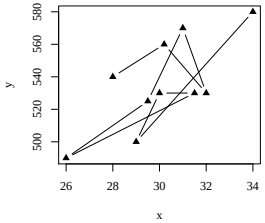
```
1  > plot(x,y,pch=17,type="l")
```

Modyfikacja grafu - kolorowanie

Zanim zaczniemy, jeden problem:

```
1  > plot(x,y,pch=17,type="l")
```

Podane przez

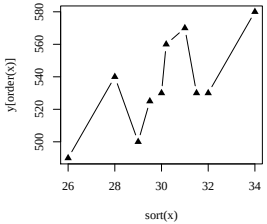


Modyfikacja grafu - kolorowanie

Zanim zaczniemy, jeden problem:

```
1  > plot(x,y,pch=17,type="l")
```

Ale chcielibyśmy

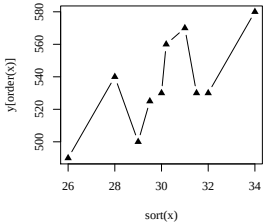


Modyfikacja grafu - kolorowanie

Zanim zaczniemy, jeden problem:

```
1  > plot(x,y,pch=17,type="l")
```

Ale chcielibyśmy



Jak to zorganizować?

Modyfikacja wykresu – styl linii

Odpowiedź

Użyj `sort()` i `order()`.

```
1  > plot(sort(x),y[order(x)],pch=17,type="b")
2  > dev.off()
```

Modyfikacja grafu - kolorowanie

Kolory możemy modyfikować za pomocą

- przez nazwy koloru elementu, na przykład `col=red`.
- numer koloru elementu, na przykład `col=636`
- według kodu szesnastkowego (w trybie RGB), na przykład `col="#FFCC00"`.

Listę dostępnych kolorów można uzyskać na wyjściu funkcji `colors()`.

Modyfikacja grafu - kolorowanie

Spróbujmy

```
> plot(sort(x),y[order(x)],pch=17,type="b",col="red")
> dev.off()
> plot(sort(x),y[order(x)],pch=17,type="b",col=636)
> dev.off()
> plot(sort(x),y[order(x)],pch=17,type="b",col="#FFCC00")
> dev.off()
```

Modyfikacja grafu - kolorowanie

Inne opcje kolorystyczne to

`col.axis`	Kolor etykiet osi.	`col.lab`	Kolor oznaczeń osi.
`col.main`	Kolor nagłówka głównego	`col.sub`	Kolor podtytułu
`bg`	Kolor wypełnienia znaków.	`fg`	Kolor tła wykresu.

Modyfikacja wykresu - kolorowanie

Spróbujmy

```
1 >plot (sort(x),y[order(x)],lty =1, type ="b",col="aquamarine",lwd =2,
2   col.axis ="violet",col.main ="green",main ="Main␣title",fg="red",
3   col.lab="coral3",pch=17)
4 > dev.off()
5 >par(bg="beige")
6 >plot (sort(x),y[order(x)],lty =1, type ="b",col=30,lwd =2,
7   col.axis ="darkmagenta", col.main ="blue3",col.sub="blue2",
8   main ="Main␣title", sub="Subtitle", fg="red",col.lab="coral4",pch=17)
9 > dev.off()
```

Modyfikacja grafu - kolorowanie

Kolory jako wektory

Wartość argumentu `col` może być ustawiona jako wektor.

Kolory z wektora są następnie regularnie przeplatane.

Możemy również użyć funkcji `rainbow()` z predefiniowaną sekwencją kolorów.

Modyfikacja grafu - kolorowanie

Spróbujmy

```
1  > plot(sort(x),y[order(x)],pch=17,type="b",
2    col=c("czerwony", "niebieski")
3  > dev.off()
4  >plot(sort(x),y[order(x)],pch=17,type="b",col=rainbow(5))
5  > dev.off()
```

Edytowanie ramki – tytuły i napisy

Podstawowe funkcje rysowania w R zawierają argument `main`, który umożliwia dodanie tytułu do wykresu.

Możemy również użyć argumentu `sub`, aby dodać podtytuł, który zostanie umieszczony pod wykresem.

Alternatywnym sposobem na dodanie tytułu i podtytułu do wykresu jest użycie funkcji `title()`.

Edycja wykresu – tytuły i napisy

Spróbujmy

```
> plot(sort(x),y[order(x)],pch=17,type="b",col=rainbow(4))
> title(main="Icecream␣sales",col.main="red")
> title(sub="Temperature",col.sub="blue",adj=1,line=2)
> dev.off()
```

Edycja wykresu – dodaj tekst do wykresu

Do narysowanego wykresu możemy dodać dowolny tekst za pomocą funkcji `text()` oraz `mtext()`.

Funkcja `text()` umieszcza podany tekst w dowolnym miejscu obszaru rysowania, funkcja `mtext()` umieszcza tekst na krawędziach.

Funkcja `text()` przyjmuje dwa dodatkowe argumenty:

- `location` definiuje współrzędne `x` i `y`, w których zostanie umieszczony tekst. Współrzędne muszą być podane jako dwa pierwsze argumenty funkcji.
- `pos` określa położenie względem aktualnej pozycji, 1=w dół, 2=w lewo, 3=w górę i 4=w prawo. Zdefiniowanie pozycji jako `locator(1)` pozwala na pozycjonowanie tekstu za pomocą myszy.

Edycja wykresu – dodaj tekst do wykresu

Spróbujmy

```
1 > plot(sort(x),y[order(x)],pch=17,type="b",col=30)
2 > title(main="Icecream␣sales",col.main="red")
3 > title(sub="Temperature",col.sub="blue",adj=1,line=2)
4 > text(c(28,32),c(560,500),c("Text1","Text2"),pos=1,col="red")
5 > dev.off()
```

Edycja wykresu – dodaj tekst do wykresu

Funkcja `mtext()` przyjmuje jeszcze dwa argumenty:

- `side` określa stronę obszaru wykresu, na której ma zostać umieszczona etykieta tekstowa, 1=dół, 2=lewo, 3=góra i 4=prawo.
- `line`określa numer wiersza, w którym ma zostać umieszczona etykieta. Linie są ponumerowane od 0.

Spróbujmy

```
1 > plot(sort(x),y[order(x)],pch=17,type="b",
2 +  col=30,xlab="",ylab="")
3 > mtext("Temperature",side=1,line=2,adj=1)
4 > mtext("Sales",side=2,line=2)
```

Edycja wykresu – wyrównanie osi

Aby usunąć ramkę wykresu, ustaw argument funkcji rysowania `axes=FALSE`.

Dodaj nowe osie za pomocą funkcji `axes()`.

Argument funkcji `osie()` określa stronę wykresu, do której zostanie dodana oś.

Jak zwykle, liczby określają boki: 1=dolny, 2=lewy, 3=górny i 4=prawy.

Spróbujmy

```
1 > plot(sort(x),y[order(x)],pch=17,type="b",col=30,axes=FALSE)
2 > axis(1)
3 > axis(2)
```

Edycja wykresu – ustawienia osi

Inną możliwością dostosowania jest zmiana kolorów osi. Można to zrobić poprzez ustawienie opcjonalnych argumentów funkcji `axis()`:

- verb+col+ Określa kolor osi,
- `col.ticks` określa kolor punktów podziału,
- `col.axis` określa kolor kresek.

Spróbujmy

```
1 > plot(sort(x),y[order(x)],pch=17,type="b",col=30,axes=FALSE)
2 > axis(1,col="blue",col.ticks="red",col.axis=555)
3 > axis(2,col="deepskyblue2",col.ticks=444,col.axis="red")
```

Dalsze ustawienia osi

Możemy również:

- Określ liczbę znaczników o podanych wartościach początkowych i końcowych,
- aby dostosować długość i orientację znaków,
- obracać etykiety znaczników,
- Dostosuj opisy tagów,
- Usuń tagi,
- dodać mniejsze tagi za pomocą `Hmisc` .

Dalsze dostosowanie osi – rozkład interwałowy

Argumenty `xaxp` i `yaxp` umożliwiają ustawienie położenia znaczników podziału na osiach `x` i `y`.

Ich wartości są ustawiane jako wektory `c(start,end,regions)`, `start` i `end` definiują wartości początkowe i końcowe na każdej osi, a `regions` określa liczbę interwałów, na które podzielona jest oś.

```
1 > plot(sort(x),y[order(x)],pch=17,type="b",col=30,axes=FALSE)
2 > axis(1,col="blue",col.ticks="red",col.axis=555,xaxp=c(26,34,8))
3 > axis(2,col="blue",col.ticks="red",col.axis=555,yaxp=c(490,580,9))
```

Dalsze ustawienia osi – długość i orientacja markerów

.

Argument `tck` pozwala na ustawienie długości i orientacji znaczników podziału.

Jego dodatnia wartość orientuje znaki wewnątrz obszaru rysowania, podczas gdy wartości ujemne orientują znaki poza obszarem rysowania. Im większa jest wartość bezwzględna, tym dłuższe są znaki. Domyślną wartością jest `tck=-0.05`.

Obrót jest włączany przez argument `las`, który może przyjąć jedną z czterech wartości:

- `las=0` etykiety są równoległe do osi (domyślnie),
- `las=1` wszystkie etykiety są poziome,
- `las=2` etykiety są prostopadłe do osi,
- `las=3` wszystkie etykiety są pionowe.

Dalsze ustawienia osi - długość i orientacja markerów

.

Spróbuj

```
> plot(sort(x),y[order(x)],pch=17,type="b",col=30,axes=FALSE)
> axis(1,col="blue",xaxp=c(26,34,8),tck=0.02,las=3)
> axis(2,col="blue",yaxp=c(490,580,9),tck=0.02,las=2)
```

Uwaga

Możemy całkowicie usunąć separatory, ustawiając argumenty `xaxt="n "` dla osi `x` lub `yaxt="n "` dla osi `y`.

Dalsze ustawienia osi – opis znaczników podziału

Opisy etykiet podziału można zmienić za pomocą argumentu `labels` funkcji `axis()`.

Aby poprawnie rozmieścić etykiety, musimy ustawić ich pozycję za pomocą argumentu `at`.

```
1  > plot(sort(x),y[order(x)],pch=17,type="b",col=30,axes=FALSE)
2  > axis(1,col="blue",at=seq(round(min(x)),round(max(x)),by=1),
3  +labels=0:8)
4  > axis(2,col="blue",yaxp=c(490,580,9),tck=0.02,las=2)
```

Zakres osi i dostosowanie do potrzeb użytkownika

Zakres wartości dla osi można zdefiniować za pomocą opcjonalnych argumentów `xlim` i `ylim` funkcji `plot()`.

Granice są określone jako wektory postaci `c(start,end)`.

Możemy również przekształcić osie do skali logarytmicznej, ustawiając argument `log` na wartość osi, którą planujemy dopasować.

`log="x "` ustawia skalę logarytmiczną na osi `x`,
`log="y "` ustawia skalę logarytmiczną na osi `y`, oraz
`log="xy "` przekształca obie osie na skalę logarytmiczną.

Zakres osi i dostosowanie do potrzeb użytkownika

Spróbuj

```
> plot(sort(x),y[order(x)],pch=17,type="b",col=30,axes=FALSE,
+ ylim=c(400,600))
> axis(1,col="blue",at=seq(round(min(x)),round(max(x)),by=1),
+ labels=0:8)
> axis(2,col="blue",yaxp=c(490,580,9),tck=0.02,las=2)
```

Podwójna oś pionowa

Príklad.
Chcemy nanieść na jeden wykres dwie charakterystyki stanu zdrowia pacjentów, temperaturę i ciśnienie krwi.

Podwójna oś pionowa

Príklad.
Chcemy nanieść na jeden wykres dwie charakterystyki stanu zdrowia pacjentów, temperaturę i ciśnienie krwi.

W zmiennych `y` i `z` mamy dane dotyczące 100 pacjentów, a zmienna `x` zawiera ciąg identyfikatorów pacjentów, liczb od 1 do 100.

Podwójna oś pionowa

Najpierw dopasuj krawędzie obszaru rysowania za pomocą `par(mar = c(3, 4, 2, 4))`.

Następnie wykreślamy wykres punktowy zmierzonych temperatur.

Ważnym krokiem jest ustawienie nowego grafu za pomocą `par(new=TRUE)`. Teraz jesteśmy gotowi do wykreślenia drugiego zestawu danych w kolorze niebieskim, bez ramek i bez osi.

Podwójna oś `y` jest wykreślana za pomocą funkcji `axis(4)` po prawej stronie wykresu.

Podwójna oś pionowa

```
x<-1:100 # generowanie danych
y <- runif(100, min=35, max=40)
z <- y+10*runif(100,min=7,max=12)
par(mar = c(3, 4, 2, 4))
plot(x, y, pch = 19, ylab = "Temperature")
par(new=TRUE)
plot(x, z, col = 4, pch = 19,
     axes = FALSE, # Brak osi
     bty = "n", # Brak ramki
     xlab = "", ylab = "")
axis(4)
mtext("Blood preasure", side = 4, line = 3, col = 4)
```

Krzywe

Jedną z wielu przydatnych funkcji w R jest `curve()`.

Jest to poręczna, mała funkcja, która pozwala na wykreślanie krzywych, takich jak wykresy funkcji.

Funkcja `curve()` przyjmuje jako pierwszy argument wyrażenie w składni R.

Na przykład,

```
curve(x^2)
curve(x^2,xlim=c(-2,2),col="red",lwd=2)
```

Wyświetl dwie lub więcej krzywych na jednym wykresie

Używamy funkcji `curve()` z argumentem `add=TRUE`.

Na przykład

```
curve(x^2)
curve(sqrt(x),col="red",lwd=2,add=TRUE)
```

Wyświetl dwie lub więcej krzywych na jednym wykresie

Użycie funkcji `curve()` nie jest ograniczone do jego samodzielnego użycia.

Możliwe jest wykreślenie pewnych danych, a następnie użycie funkcji `curve()` do narysowania nad nimi dowolnej linii.

```
1  set.seed(1)
2  x <- rnorm(100)
3  y <- x^2 + rnorm(100)
4  plot(y ~ x)
5  curve(x^2,add=TRUE)
```

Dodawanie legendy

Funkcja legenda()+ pozwala na dodanie legendy do wykresów.

Kilka argumentów:

- verb+x,y+ pozycja w obszarze rysowania określona przez współrzędne na wykresie,
- `legend` wektor ciągów znaków dla opisu legendy,
- `col` wektor kolorów użytych na wykresie,
- `pch` wektor kształtów markerów użytych w wykresie,
- `lty` wektor typów linii użytych w wykresie,
- `ncol` liczba kolumn używanych w legendzie, wartość domyślna to jedna kolumna.

Uzupełnienie do legendy

Príklad.
Utwórzmy funkcję użytkownika `gonplot()`, która wykreśla wykresy $\sin x$ i $\cos x$ w zakresie $(-10; 10)$ w dwóch kolorach i różnymi typami linii. Następnie dodajemy legendę do wykresu.

Uzupełnienie do legendy

Funkcja zdefiniowana przez użytkownika

```
1 gonplot <- function() {
2     curve(sin(x),xlim=c(-10,10),col="red",lwd=2,type="l",
3     ylab="sin␣x",xlab="",ylim=c(-1,2))
4     curve(cos(x),xlim=c(-10,10),col="blue",lwd=2,type="l",lty=2,
5     ylab="sin␣x",xlab="",add=TRUE)
6 }
```

Uzupełnienie do legendy

Wyświetlanie wykresu i dodawanie legendy

```
1 gonplot()
2 legend(x="topright", # Pozycje
3        legend=c("sin␣x", "cos␣x"), # tekst legendy
4        lty = c(1, 2), # Typ linii
5        col = c("red", "blue"), # Kolory linii
6        lwd = 2)
```

Uzupełnienie do legend - notatka

Argument `x position` może być ustawiony na jedną z wartości:

`top`, `topleft`, `topright`, `bottom`, `bottomleft`, `bottomright`, `left`, `right` lub `center`.

Ten scenariusz nie wymaga ustawiania argumentu `y`, ponieważ pozycja legendy jest określona słownie.

Wykresy słupkowe

Wykres słupkowy wyświetla dane kategoryczne za pomocą prostokątnych kolumn, których wysokość lub długość jest proporcjonalna do reprezentowanych przez nie wartości.
R używa funkcji do tworzenia wykresów słupkowych

```
barplot(H,xlab,ylab,title, names.arg,col)
```

Parametry wykorzystywane w funkcji są następujące:

- `H` jest wektorem lub macierzą zawierającą wartości liczbowe używane w wykresie słupkowym,
- `xlab` to etykieta osi x,
- `ylab` jest etykietą osi y,
- `title` to tytuł wykresu słupkowego,
- `names.arg` to wektor etykiet, które pojawiają się pod każdą kolumną,
- `col` służy do przypisywania kolorów kolumnom na wykresie.

Wykresy słupkowe

Załóżmy, że wektor `x` zawiera dzienną sprzedaż pewnych produktów. Wielkość sprzedaży można przedstawić w postaci wykresu słupkowego.

```
1  x<-c(2000,2400,1400,2600)
2  barplot(x)
```

Wykresy słupkowe – słupki poziome

Ustaw argument `horiz=T` na true.

```
1 barplot(x,horiz=T)
```

Wykresy kolumnowe – kolorowanie i etykiety kolumn

Używamy parametru `names.arg` wykresu słupkowego, aby przypisać nazwy do kolumn.

Następnie określamy wartości parametrów

- `xlab` a `ylab` dla nazw osi,
- `col` a `border` do kolorowania pasków, a także
- `main` aby zdefiniować tytuł wykresu.

Jest to funkcja podobna do funkcji `plot()`.

Wykresy kolumnowe – kolorowanie i etykiety kolumn

Niech nasz wektor x reprezentuje dzienną sprzedaż pewnego owocu.

Ustawiamy ich nazwy jako wektor goods i używamy go do przypisania nazw do kolumn.

```
1 goods<-c("orange","banana","apple","plum")
2 barplot(x,names.arg=goods,xlab="Fruit",ylab="Sales",
3 col="cyan",main="Monthly␣sale",border="black")
```

Wykresy kolumnowe – kolorowanie i etykiety kolumn

Możemy dostosować wykres z różnymi kolorami kolumn

Ustawiamy pożądane kolory jako wektor `colours` i używamy go jako wartości argumentu `col`.

Argument `border` określa kolor obramowania kolumn.

```
1 colours<-c("orange","yellow","red","blue")
2 barplot(x,names.arg=goods,xlab="Fruit",ylab="Sales",
3 col=colours,main="Monthly␣sale",border="black")
```

Wykresy słupkowe – z proporcjami

Używając macierzy wartości wejściowych, możemy oznaczyć kolorem i etykietą proporcje w kolumnach.

Zakres wielkości sprzedaży w wektorze `x` w ciągu wielu miesięcy.

Następnie przedstawiamy te informacje w formie graficznej.

Po pierwsze, ustawiamy

```
1 months<-c("Jan", "Feb", "Mar", "Apr")
2 x<-matrix(c(2000,2400,1400,2600,1800,2200,1600,2400,2100,
3 2300,1500,2400,2400, 1800,1200,2200),nrow=4,ncol=4)
```

wykresy słupkowe – z proporcjami

Teraz jesteśmy gotowi narysować wykres z udziałami proporcjonalnymi

Dodamy również legendę

```
1 barplot(x, main = "Sale volumes", names.arg = months,
2 xlab = "Month", ylab = "Sales",
3 col = colours, ylim=c(0,11000))
4 legend("topright", goods, fill = colours,ncol=2)
```

Wykresy słupkowe – z proporcjami

Te same informacje możemy również przedstawić w formie wykresu grupując kolumny.

Ustaw argument beside=T.

```
1 barplot(x, beside=T,main = "Sale volumes",
2 names.arg = months, xlab = "Month", ylab = "Sales",
3 col = colours, ylim=c(0,3000))
4 legend("topright", goods, fill = colours,ncol=2)
```

Wykresy słupkowe – wypełnienie teksturą

Zamiast kolorami, możemy wypełnić paski teksturami.

Linie równoległe są najprostsze

Gęstość linii możemy kontrolować za pomocą argumentu `density`, którego wartością jest wektor o długości równej liczbie linii.

Podobnie możemy określić kąt nachylenia linii wypełnienia, podając argument `angle` jako wektor, którego długość jest równa liczbie kolumn.

Wykresy słupkowe - wypełnienie liniami

```
1  x<-c(2000,2400,1400,2600)
2  barplot(x,density=c(5,10,20,30), angle=c(0,30,60,90),
3  col="blue",names.arg=goods,main="Sale volumes",
4  xlab="Fruit",ylab="Sales")
```

Wykresy słupkowe – wypełnianie przez przecinanie linii

```
1  angle1<-c(0,30,60,90)
2  angle2<-c(90,120,150,0)
3  barplot(x,density=c(10,15,20,25), angle=angle1,beside = TRUE,
4   main="Sale␣volumes", col = colours,names.arg=months, xlab = "Month",
5   ylab = "Sales",ylim=c(0,3000))
6  barplot(x,density=c(10,15,20,25), angle=angle2,beside = TRUE,
7  col = colours, add=TRUE)
8  legend("topright", goods, ncol=2, fill=colours, angle=angle1,
9   density=c(10,15,20,25))
10 legend("topright", goods, ncol=2, fill=colours, angle=angle2,
11  density=c(10,15,20,25))
```

Histogramy

Histogram to przedstawienie przybliżonego rozkładu danych liczbowych.

Pokazuje częstości występowania wartości podzielonych na przedziały.

Histogram jest podobny do wykresu słupkowego, z tą różnicą, że grupuje wartości w ciągłe przedziały.

Histogramy dają przybliżony obraz gęstości rozkładu danych.

Histogramy

Histogram można utworzyć za pomocą funkcji `hist()` w programie R.

```
barplot(H,xlab,ylab,title, names.arg,col)
```

Parametry wykorzystywane w funkcji są następujące:

- `data` to wektor zawierający wartości numeryczne używane w histogramie,
- `main` to tytuł wykresu,
- `col` służy do ustawiania koloru pasków,
- `border` służy do ustawiania koloru obramowania każdej kolumny,
- `xlab` określa opis osi x,
- `xlim` określa zakres wartości na osi x,
- `ylim` określa zakres wartości na osi y,
- `breaks` służy do określenia szerokości każdej kolumny.

Histogramy - przykład

Príklad.
Zilustrujmy histogramy na przykładzie rzutu kostką do gry. Załóżmy, że rzucamy dwiema kostkami 10 000 razy i interesuje nas suma wyrzuconych punktów.

Histogramy - przykład

Po pierwsze, symulujemy rzut kostką:

```
1 dice1<-sample(1:6,replace=T,10000)
2 dice2<-sample(1:6,replace=T,10000)
3 c<-dice1+dice2
```

Teraz możemy wyświetlić histogram sum za pomocą funkcji `hist()`:

```
1 hist(c,breaks=1.5:12.5, main="Rolling␣2␣dice",
2 xlab="two␣dice", ylab="Frequency")
```

Histogramy - przykład

Centralne twierdzenie graniczne, znane z teorii prawdopodobieństwa, stwierdza, że w wielu sytuacjach, w których niezależne zmienne losowe są dodawane razem, ich średnia suma zbiega do rozkładu normalnego (nieformalnie, krzywej dzwonowej), nawet jeśli oryginalne zmienne same nie są normalnie rozłożone.

Można to udokumentować na histogramie, wykreślając krzywą gęstości rozkładu normalnego na tym samym wykresie co histogram.

```
1 hist(c,breaks=1.5:12.5, main="Rolling␣2␣dice",
2 xlab="two␣dice", ylab="Frequency")
3 curve(dnorm(x,mean(c),sd(c))*10000,col="red",add=T)
```

Diagramy kołowe

Diagram kołowy jest wykresem dla jednej zmiennej kategorycznej i jest alternatywą dla wykresu słupkowego.

Diagram kołowy (lub wykres/diagram tortowy) to wykres w kształcie koła podzielony na części, które pokazują zależności między zmiennymi.

W wykresie kołowym długość łuku każdego segmentu (a zatem jego kąt środkowy i powierzchnia) jest proporcjonalna do wielkości, którą reprezentuje.

Diagramy kołowe

Podstawowa składnia do tworzenia wykresu kołowego w R jest następująca:

```
pie(data, labels, radius, main, col, clockwise)
```

Znaczenie argumentów:

- `data` jest wektorem zawierającym wartości liczbowe użyte w wykresie kołowym,
- `labels`Do opisania części używa się przysłówka + znaków+,
- `radius` określa promień diagramu kołowego (wartość pomiędzy -1 a $+1$),
- verb+main+
- wskazuje tytuł wykresu,
- `col` wskazuje paletę kolorów,
- `clockwise` jest to wartość logiczna określająca, czy plasterki są rysowane zgodnie czy przeciwnie do ruchu wskazówek zegara.

Diagram kołowy - przykład

Príklad.
Załóżmy, że chcemy przedstawić udziały w miesięcznych wydatkach gospodarstwa domowego za pomocą wykresu kołowego. Bierzemy pod uwagę następujące kategorie wydatków: mieszkanie, żywność, odzież, rozrywka i inne.

Wartości, które wykorzystujemy jako parametry wykresu kołowego:

```
1 data<-c(200,300,100,80,150)
2 labels<-c("housing","food","clothing","entertainment","other")
3 pie(data,labels,main="Monthly␣expenses")
```

Diagram kołowy – regulacja koloru

Aby zmienić kolory na wykresie, używamy funkcji `rainbow()`, która definiuje paletę kolorów. Jego argumenty są następujące:

- `n` liczba kolorów (*geq*1), które mają być w palecie,
- `s,v` "nasycenie koloru" i "wartośći", aby dodać opisy do kolorów
- `start` (zmodyfikowany) odcień w $\langle 0; 1\rangle$, od którego zaczyna się wybrana tęcza,
- `end` (dostosowany) odcień w miejscu, gdzie kończy się tęcza,
- `gamma` Korekcja gamma dla każdego koloru (r,g,b)+ w przestrzeni RGB (ze wszystkimi wartościami w przestrzeni $\langle 0; 1\rangle$), kolor wynikowy odpowiada $(r^\gamma, g^\gamma, b^\gamma)$,
- `alpha`transparency, liczba w postaci $\langle 0; 1\rangle$, (0 oznacza przezroczyste, a 1 oznacza nieprzezroczyste).

Diagram kołowy - użycie rainbow()

```
1 description<-paste(labels,"\n",data,sep="")
2 pie(data,description,main="Monthly␣expenses",
3 col=rainbow(length(data)))
```

Uwaga

Zmieniliśmy również etykiety. Do ich nazw dodaliśmy wartości liczbowe.

Diagram kołowy – dalsze ulepszenia

Jako kolejne ulepszenia możemy wymagać opisów z procentami oraz wyświetlania wykresów z efektem 3D.

Najpierw musimy przeliczyć procenty i dodać wyniki do opisów. Aby otrzymać procenty jako liczby całkowite, używamy funkcji `trunc()`.

Następnie możemy stworzyć wykres kołowy (tym razem korzystając z palety) `heat.colors()`.

```
1 description<-paste(labels,"\n",trunc(100*data/sum(data)),
2 "%", sep="")
3 pie(data,description,main="Monthly␣expenses",
4 col=heat.colors(length(data)))
```

Diagram kołowy – dalsze ulepszenia

Aby uzyskać efekt 3D na wykresie, należy użyć pakietu `plotrix`.

Używamy `pie3D()` do tworzenia wykresów z efektem 3D.

```
1 library("plotrix")
2 pie3D(data,labels=description,main="Monthly␣expenses",
3  col=rainbow(length(data)))
```

Diagram kołowy – rozdzielenie części

Wygląd wykresu 3D możemy dodatkowo dostosować za pomocą parametrów

- `height`, który określa wysokość tortu 3D (domyślna wartość to 0.1)
- `theta`, który zmienia kąt widzenia (domyślna wartość to *fracpi*6).
- `explode`, który definiuje podział części tortu

```
1 pie3D(data,labels=description,main="Monthly␣expenses",
2  col=terrain.colors(length(data)),height=0.2,theta=1.5,
3 explode=0.1)
```

Uwaga na użycie palety `terrain.colors`

Wachlarzowy diagram

Użyteczną alternatywą dla wykresów kołowych jest funkcja `fan.plot()` zdefiniowana w pakiecie `plotrix`.

Umożliwia wizualne porównanie sektorów wykresu kołowego.

Wykres wachlarzowy może być dostosowany do potrzeb użytkownika poprzez podanie dodatkowych argumentów:

- `max.span` Kąt maksymalnej rozpiętości sektora w radianach. Domyślnie, `data` jest skalowane tak, aby suma była równa $2pi$.
- `ticks` liczba łatek, które pojawiłyby się, gdyby sektory znajdowały się na wykresie kołowym. Domyślnie nie ma skrzynek.

Diagram wachlarzowy

Ilustracja wykresu wachlarzowego

```
1 fan.plot(data,labels=description,main="Monthly␣expenses",
2  col=rainbow(length(data)),max.span=pi,ticks=max(data))
```

Diagram wachlarzowy

Ilustracja wykresu wachlarzowego

```
1 fan.plot(data,labels=description,main="Monthly␣expenses",
2  col=rainbow(length(data)),max.span=pi,ticks=max(data))
```

Wadą wykresu wachlarzowego jest duża biała przestrzeń nad wykresem.

Miejsce to można usunąć poprzez ustawienie nowego urządzenia graficznego o wysokości i szerokości zdefiniowanej przez użytkownika.

Otwieramy nowe okno graficzne za pomocą funkcji `new.dev()`. Określamy rozmiar okna za pomocą argumentów `height` i `width`.

Diagram wachlarzowy

```
1  dev.new(width=10,height=5,unit="cm")
2  fan.plot(data,labels=description,main="Miesieczne␣wydatki",
3   col=rainbow(length(data)),max.span=pi,ticks=max(data))
```

Wykres pudełkowy

Boxploty są tworzone w R za pomocą funkcji `boxplot()`. Podstawowa składnia do tworzenia boxplot w R to:

```
boxplot(x, data, notch, varwidth, names, main)
```

Znaczenie parametrów jest następujące:

- `x` jest wektorem lub formułą,
- `data` jest ramką danych.
- `notch` jest wartością logiczną. Ustawienie TRUE powoduje narysowanie wcięcia.
- `varwidth` jest wartością logiczną. Ustawiona jako prawda powoduje rysowanie szerokości pudełka proporcjonalnej do rozmiaru próbki,
- `names` są etykietami grup, które będą drukowane pod każdym wykresem,
- `main` jest używany do nadania tytułu wykresowi.

Wykres pudełkowy - przykład

Príklad.
Załóżmy, że mamy statystyki z meczu koszykówki w pliku danych `players.csv`. Ten plik danych zawiera identyfikatory zawodników, ich pozycję, liczbę prób strzałów i liczbę udanych prób strzałów. Używając boxplotów, porównajmy punkty zdobyte na każdej pozycji.

```
1 players<-read.csv("players.csv")
2 boxplot(made~position,data=players,
3 xlab="Position",ylab="Points␣gained",
4 main="Scoring␣by␣position")
```

Wykres pudełkowy

Podobnie jak w przypadku innych typów wykresów, możemy dostosować wygląd wykresu.

Ilustrujemy kolorowanie wykresu i dostosowujemy szerokość ramek tak, aby była proporcjonalna do liczebności próby, ustawiając `varwidth=TRUE`.

```
1 boxplot(made~position,data=players,
2 xlab="Position",ylab="Points␣gained",
3 main="Scoring␣by␣position",col="cyan",varwidth=TRUE)
```

Wykres pudełkowy

Ustawiając zmienną logiczną `horizontal` na `TRUE` możemy obracać wykresy boxplotu.

Ponadto, kolory mogą się różnić w zależności od boxplotu.

```
1 boxplot(made~position,data=players,
2 xlab="Position",ylab="Points␣gained",
3 main="Scoring␣by␣position",
4 col="col=c("blue","cyan","green"),
5 varwidth=TRUE,horizontal=TRUE)
```

Wykresy Q-Q

Wykres kwantylowy (w skrócie wykres Q-Q) jest narzędziem graficznym, które pomaga nam ocenić, czy zbiór danych wiarygodnie pasuje do pewnego teoretycznego rozkładu, takiego jak rozkład normalny lub wykładniczy.

Na przykład, jeśli przeprowadzamy analizę statystyczną, która zakłada, że nasza zmienna zależna ma rozkład normalny, możemy użyć normalnego wykresu Q-Q, aby sprawdzić to założenie.

Jest to tylko wizualna kontrola, a nie dokładny dowód, ale pozwala nam zobaczyć na pierwszy rzut oka, czy nasze założenie jest wiarygodne, a jeśli nie, to w jaki sposób założenie jest naruszone i które wartości danych przyczyniają się do naruszenia.

Wykresy Q-Q

Wykres Q-Q jest zasadniczo wykresem punktowym utworzonym przez wykreślenie dwóch zestawów kwantyli względem siebie.

Jeśli oba zestawy kwantyli pochodzą z tego samego rozkładu, punkty leżą w przybliżeniu na linii prostej.

Wykresy Q-Q biorą nasze próbki, sortują je w porządku rosnącym, a następnie wykreślają je względem kwantyli proponowanego rozkładu teoretycznego.

Liczba kwantyli jest tak dobrana, aby odpowiadała wielkości naszej próby.

Wykresy Q-Q

W R mamy dwie funkcje do tworzenia Q-Q graphs:

`qqnorm()` tworzy normalny wykres Q-Q (co oznacza, że proponowany teoretyczny rozkład jest normalny),

`qqplot()` pozwala na utworzenie wykresu Q-Q do porównania dwóch zestawów danych.

Związana z funkcją `qqnorm()` jest funkcja `qqline()`, która kreśli linię kwantyla normalnego, która przechodzi przez kwantyl "probs", standardowy pierwszy i trzeci kwartyl.

Wykresy Q-Q – Ilustracja

Najpierw tworzymy próbkę z rozkładu normalnego

W kolejnym kroku porównujemy go z rozkładem teoretycznym

```
1  x<-rnorm(100,mean=10,sd=1)
2  qqnorm(x)
3  qqline(x, col="steelblue", lwd=2)
```

Wykresy Q-Q – Ilustracja

Aby zobrazować sytuację, w której próbka nie pochodzi z założonego rozkładu, wygenerujemy próbkę z rozkładu wykładniczego.

```
1 x<-rexp(100,rate=1/10)
2 qqnorm(x)
3 qqline(x, col="steelblue", lwd=2)
```

Wykresy Q-Q – Ilustracja

Jeśli dwie próbki losowe pochodzą z tego samego typu rozkładu, to dla porównania tworzymy dwa wektory `x` i `y`.

Następnie stosujemy funkcję `qqplot()` do tych próbek.

```
1  x<-rnorm(100,mean=10,sd=1)
2  y<-rnorm(100,mean=5,sd=3)
3  qqplot(x,y,main="Q-Q␣plot␣for␣two␣samples")
```

Wykresy Q-Q – Ilustracja

Funkcja `qqplot()` nie współpracuje z `qqline()`.

Jeśli chcemy dodać linię pomocniczą, używamy funkcji `abline()` wraz z funkcją `sort()`.

```
1 x<-rnorm(100,mean=10,sd=1)
2 y<-rnorm(100,mean=5,sd=3)
3 qqplot(x,y,main="Q-Q␣plot␣for␣two␣samples")
4 abline(lm(sort(y) ~ sort(x)), col = "steelblue", lwd = 2)
```

Funkcja `lm()` tworzy model zależności liniowej i dostarcza współczynniki potrzebne do wykreślenia linii.

Wykres Q-Q

Funkcja `qqplot()` może być użyta do porównania próbki z dowolnym rozkładem teoretycznym.

Tworzymy wektor kwantyli o tej samej długości co próba i używamy tego wektora jako drugiego zestawu danych wprowadzanego do funkcji `qqplot()`.

```
1  x<-rexp(100,rate=1/10)
2  y<-qexp(seq(0,1,by=0.01),rate=1)
3  qqplot(x,y,main="exponential␣Q-Q␣plot")
4  abline(lm(sort(y[1:100]) ~ sort(x)),col="steelblue",
5   lwd = 2)
```

Wielość wykresów w jednym obrazie

W R możemy łączyć wykresy za pomocą parametrów graficznych `mfrow` i `mfcol`.

Wystarczy podać wektor określający liczbę wierszy i liczbę kolumn, które zamierzamy utworzyć.

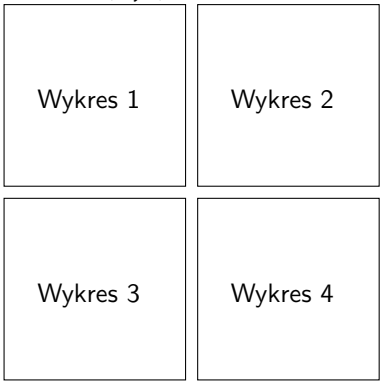
Decyzja o tym, który parametr wykresu zastosować zależy od tego, jak chcemy ułożyć wykresy:

- `mfrow` wykresy będą ułożone w rzędach,
- `mfcol` wykresy będą ułożone według kolumn.

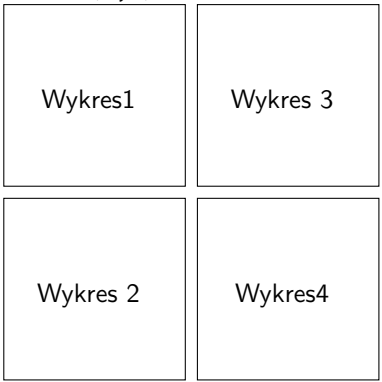
To ustawienie jest używane jako argument funkcji `par()`, która modyfikuje parametry urządzenia graficznego.

Wielokrotne wykresy w jednym obrazie

`mfrow=c(2,2)`



`mcol=c(2,2)`



Wielokrotne wykresy w jednym obrazie – ilustracja

```
set.seed(5)
x <- rexp(80)
# Dwa rzedy, dwie kolumny
par(mfrow = c(2, 2))
# Wykresy
hist(x, main = "Histogram") # Lewy gorny
boxplot(x, main = "Box plot") # Prawy gorny
plot(x, main = "Scatter plot") # Lewy dolny
pie(table(round(x)), main = "Pie chart") # Prawy dolny
# Powrot do oryginalnego urzadzenia graficznego
par(mfrow = c(1, 1))
```

Wielokrotne wykresy w jednym obrazie – bardziej złożona struktura

Często potrzebujemy stworzyć obraz o bardziej złożonej strukturze.

W takich sytuacjach musimy skorzystać z funkcji `layout()`. Funkcja ta przyjmuje cztery ważne argumenty:

- `mat` macierz, gdzie każda wartość reprezentuje lokalizację obrazów.
- `widths` wektor dla szerokości kolumn. Możemy je również podać w centymetrach za pomocą funkcji `lcm()`.
- `heights` wektor wysokości kolumn. Możemy je również podać w centymetrach za pomocą funkcji `lcm()`.
- `respect` wartość logiczna lub macierz wypełniona 0 i 1 o tych samych wymiarach co `mat`, aby określić, czy respektować relacje między szerokościami i wysokościami.

Wielokrotne wykresy w jednym obrazie – bardziej złożona struktura

Przed dodaniem wykresów możemy wyświetlić podgląd layoutu za pomocą funkcji `layout.show()`.

```
1 l <- layout(matrix(c(1, 2, 2, # Pierwszy, drugi,
2                      3, 3, 4), # trzeci i czwarty wykres
3              nrow=2,
4              ncol=3,
5              byrow=TRUE))
6 layout.show(l)
```

Wielokrotne wykresy w jednym obrazie – bardziej złożona struktura

Metodę tę ilustrujemy na wykresie punktowym zaimplementowanym z krawędziami w postaci histogramu i wykresu pudełkowego.

```
1  l <- layout(matrix(c(2, 0, 1, 3),
2                nrow = 2, ncol = 2,
3                byrow = TRUE),
4         widths = c(9, 3),
5         heights  = c(3, 9), respect = TRUE)
6  plot(x, main = "Scatter␣plot")
7  hist(x, main = "Histogram")
8  boxplot(x, main = "Box␣plot")
```

Statystyka i programowanie w R

VI. Charakterystyky doboru próby

Średnia

Musimy użyć funkcji `as.numeric()` do obliczenia średniej, ponieważ `cars[1,]` podaje wartości w formacie listy.

Aby więc uzyskać średnią wartość nowych samochodów osobowych rejestrowanych miesięcznie (w tysiącach) w latach 2017-18, użyjemy kodu:

```
1 cars<-read.csv2("macrostat.csv",header=FALSE,sep=";")
2 mean(as.numeric(cars[1,]))
3 [1] 8.090125
```

Średnia

Często zdarza się, że interesujące nas wartości o charakterze statystycznym układają się w ciąg bezwzględnych obfitości.

W tym przypadku modyfikujemy relację (??), aby obliczyć średnią wartość dla kształtu:

$$\overline{x} = \frac{x_1 \cdot n_1 + x_2 \cdot n_2 + \cdots + x_k \cdot n_k}{n_1 + n_2 + \cdots + n_k} = \frac{\sum_{i=1}^{k} x_i \cdot n_i}{\sum_{i=1}^{k} n_i}. \tag{2}$$

gdzie x_i oznaczają wartości zmiennej, a n_i ich bezwzględne liczby.

Średnia

W takim przypadku musimy zdefiniować funkcję niestandardową, aby obliczyć wartość średnią.

Wprowadzamy dwa wektory jako wartości wejściowe. Pierwszy wektor zawiera wartości uzyskane przez zmienną losową, a drugi to wektor ich częstości.

Przed wykonaniem obliczeń według relacji (2) należy sprawdzić, czy oba wektory mają taką samą długość.

Średnia

Następnie możemy zdefiniować odpowiednią funkcję `mean2()` w następujący sposób:

```
1  mean2<-function(arg1,arg2){
2      if (length(arg1)==length(arg2)){
3          s<-sum(arg1*arg2)/sum(arg2)
4      }
5      else{s<-c("Arguments␣are␣not␣of␣equal␣length")}
6      return(s)
7  }
```

Średnia

Możemy zilustrować użycie właśnie zdefiniowanej funkcji `mean2()` na zmiennej, która pobiera wartości ze zbioru $\{1, 2, \ldots, 10\}$.

Częstotliwości bezwzględne tych wartości generujemy za pomocą rozkładu Poissona.

Otrzymane wartości są przechowywane w wektorze a, a ich bezwzględne częstotliwości w wektorze b.

```
1  a<-c(1,2,3,4,5,6,7,8,9,10)
2  b<-rpois(10,20)
3  mean2(a,b)
4  5.38613861386139
```

Mediana, wartość środkowa

Aby określić medianę, w języku R zaimplementowano funkcję `median()`.

Dzięki temu kodowi możemy łatwo znaleźć medianę miesięcznej liczby nowo zarejestrowanych samochodów osobowych

```
1  > median(as.numeric(cars[1,]))
2  [1] 8.2425
```

Kwantyle

Median zdefiniowany w poprzedniej sekcji dzieli próbę na dwa równie prawdopodobne podzbiory.

Ogólnie rzecz biorąc, możemy podzielić próbkę na dowolną liczbę q jednakowo prawdopodobnych części. Te wartości są nazywane q-kwantyle a k-ty q-kwantyl dla zmiennej losowej X jest określony wzorem

$$\mathbb{P}(X < x) \leq \frac{k}{q}. \tag{3}$$

Kwantyle

Funkcja `quantile()` jest zaimplementowana w języku R w celu znalezienia kwantyli. Bez określania parametrów opcjonalnych wynikiem jest minimum próby, pierwszy kwartyl, mediana, trzeci kwartyl i maksimum próby.

Możemy to zilustrować danymi na temat COVID-19, pobranymi z oficjalnej strony słowackiego rządu https://korona.gov.sk.

```
1  data<-read.csv("https://mapa.covid.chat/export/csv",
2     header=T,sep=";")
3  > quantile(data[,4])
4     0%    25%    50%    75%   100%
5      0     30    232   1737  15278
6  >
```

Kwantyle

Możemy również przekazać kilka opcjonalnych argumentów do funkcji `quantile()`:

- `probs` numeryczny wektor prawdopodobieństw o wartościach w $\langle 0, 1 \rangle$, który określa poziomy prawdopodobieństwa dla żądanych kwantyli,
- `na.rm` boolean jeśli TRUE wszystkie NA i NaN są usuwane z `data` przed obliczeniem kwantyli,
- `names` boolean, jeśli TRUE, wynik ma atrybut names. Ustaw `FALSE` dla przyspieszenia w wielu `probs`.

Kvantily

Zilustrujemy to wyznaczając decyle dziennych przyrostów

```
> quantile(data[,4],probs=seq(0,1,by=0.1))
    0%   10%   20%   30%   40%   50%   60%   70%   80%   90%  100%
     0     6    20    43    91   232   642  1293  2034  3041 15278
>
```

Rozstęp

Dane wyjściowe funkcji `range()` w środowisku języka R to zakres wariacji.

Jego wyjście to dwie wartości - największa i najmniejsza wartość w próbce.

Aby wyrazić zakres odchylenia jako z definicji jedną wartość (**??**), używamy funkcji `max()` i `min()`.

Rozstęp

Przykładowy kod źródłowy

```
1  > x<-c(5,10,12,4,16,8,9)
2  > range(x)
3  [1]  4 16
4  > R<-max(x)-min(x)
5  > R
6  [1] 12
7  >
```

Rozstęp międzykwartylowy

W języku R jest ona zaimplementowana jako funkcja `IQR()`

```
1  > x<-c(5,10,12,4,16,8,9)
2  > IQR(x)
3  [1] 4.5
4  >
```

Średnie odchylenie bezwzględne

W języku R jest ona zaimplementowana jako funkcja `mad()`

```
1  > x<-c(5,10,12,4,16,8,9)
2  > mad(x)
3  [1] 4.4478
4  >
```

Wariancja i odchylenie standardowe

Funkcji `var()` i `sd()` należy używać ostrożnie.

Ich wynikiem są obiektywne oszacowania wariancji i odchylenia standardowego całej populacji.

Jeśli chcemy obliczyć wariancję próbki zgodnie z relacją (??), musimy zdefiniować własną funkcję, którą zilustrujemy w poniższym kodzie źródłowym.

Wariancja i odchylenie standardowe

```
> variance<-function(x) sum((x-mean(x))^2)/length(x)
> stdev<-function(x) sqrt(variance(x))
> variance(x)
[1] 14.40816
> stdev(x)
[1] 3.795809
> var(x) # porownaj wyniki
[1] 16.80952
> sd(x)
[1] 4.099942
```

Współczynnik zmienności

Współczynnik zmienności jest statystyczną miarą względnego rozrzutu danych w stosunku do wartości średniej.

Współczynnik zmienności CV jest zdefiniowany jako stosunek odchylenia standardowego s do średniej $\overline{x}$

$$CV = \frac{s}{\overline{x}}. \tag{4}$$

Współczynnik zmienności jest często wyrażany w procentach.

Współczynnik zmienności

Współczynnik zmienności nie jest zaimplementowany jako funkcja w języku R

Możemy to obliczyć za pomocą znanych funkcji lub zdefiniować własną funkcję

```
1 > cv<-function(x) variance(x)/mean(x) * 100
2 > cv(x)
3 [1] 157.5893
```

Skośność

Skośność jest miarą asymetrii rozkładu lub zbioru danych.

Skośność γ_1 definiujemy jako

$$\gamma_1 = \frac{\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^3}{s^3}. \tag{5}$$

Skośność i ostrość

Potrzebujemy pakietu `moments`, aby obliczyć skośność i ostrość w R.

Funkcje `skewness()` i `kurtosis()` są zdefiniowane w tym pakiecie.

```
1  > library(moments)
2  > skewness(x)
3  [1] 0.3598295
4  > kurtosis(x)
5  [1] 2.252963
6  >
```

Statystyka i programowanie w R

VII. Szacunki parametrów

Szacunki punktowe

Metody

W tym kursie przedstawimy dwie metody konstruowania Szacunków punktowych:

- metoda momentów,
- metoda największej wiarygodności.

Załóżmy, że mamy próbkę $X_1, \ldots, X_n$ z rozkładu, który zależy od wektora parametrów $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_m)$.

Przedziały ufności

Example

Załóżmy, że przeprowadzana jest ankieta wśród 250 losowo wybranych osób w celu ustalenia, czy posiadają one tablet. Spośród 250 respondentów 98 stwierdziło, że posiada tablet. Korzystając z poziomu ufności 95 %, oblicz szacunkowy przedział ufności dla prawdziwego odsetka osób posiadających tablet.

Przedziały ufności

Rozwiązanie: Najpierw obliczamy nieobciążone oszacowanie punktowe prawdopodobieństwa p jako $\hat{p} = \frac{98}{250}$ i ustawiamy $\hat{q} = 1 - \hat{p}$.

Możemy teraz obliczyć granice przedziału ufności za pomocą funkcji `qnorm()`.

Przedziały ufności

```
> n<-250
> p<-98/n
> q<-1-p
> c<-qnorm((1+alpha)/2,0,1)
> lower.bound<-p-c*sqrt(p*q/n)
> upper.bound<-p+c*sqrt(p*q/n)
> print(c(lower.bound,upper.bound))
[1] 0.3314836 0.4525164
```

Otrzymaliśmy więc 95 % przedział ufności $(0,3315; 0,4525)$ dla odsetka osób posiadających tablety.

Dziękujemy za uwagę.

Statystyka i programowanie w R

Aleš Kozubík