

Az R nyelv

Jeszenszky Péter
Debreceni Egyetem, Informatikai Kar
jeszenszky.peter@inf.unideb.hu

2019. június 18.



1

A dim attribútum (2)

- A `dim(x)` függvény használható a `dim` attribútum értékének lekérdezésére.
- A `dim(x) <- érték` kifejezés beállítja a `dim` attribútum értékét.
 - *érték* kötelezően olyan nemnegatív elemekből álló egész vektor, melyek szorzata egyenlő `x` hosszával.
 - Megadható valós vektor is, mely egészszé lesz konvertálva.
 - Akkor megengedett 0 az elemek között, ha `x` hossza is 0 (így teljesül a feltétel).
 - Az attribútum értékének törléséhez `NULL`-t kell megadni.

3

A dim attribútum (1)

- Tömbök és mátrixok megvalósítására szolgál.
- Az attribútum értéke adja meg tömbök illetve mátrixok kiterjedését az egyes dimenziókban.
 - Értéke egy megfelelő számú nemnegatív értékből álló egész vektor.
 - Mátrixoknál speciálisan egy kételemű vektor (az első elem a sorok, a második az oszlopok száma).
- Az attribútumot a tömböket és mátrixokat manipuláló függvények, operátorok automatikusan kezelik.

2

Tömbök és mátrixok

- Megvalósítás a `dim` attribútum segítségével.
- A mátrixok kétdimenziós tömbök.
 - A `dim` attribútum értéke mátrixoknál kételemű vektor.
- A legtöbb függvény és operátor tömbökre és mátrixokra alkalmazása elemenkénti végrehajtást jelent, az eredmény is tömb illetve mátrix.
 - Vannak azonban speciálisan mátrixok kezelésére szolgáló függvények, operátorok.
- Nem csupán vektorokból, hanem akár listákból is létre lehet hozni tömböket és mátrixokat.

4

Tömbök és mátrixok létrehozása (1)

- Tömb és mátrix létrehozható egy vektorból a `dim` attribútum megadásával.
 - Tömb feltöltése során először az első index futja be az összes lehetséges értéket, ...
 - Ennek speciális eseteként mátrix feltöltése a vektor elemeivel oszlop-folytonosan történik.
- Egy tömb vagy mátrix alakja egyszerűen megváltoztatható a `dim` attribútum értékének megváltoztatásával.
- Törölve egy tömb vagy mátrix `dim` attribútumát visszkapjuk az alapul szolgáló vektort.

5

Tömbök és mátrixok létrehozása (3)

- Kényelmesebben létrehozhatunk tömböket és mátrixokat az `array()` és `matrix()` függvényekkel.
 - Ezeknél az alapul szolgáló vektor hossza kisebb is lehet a kívánt mérethez szükségesnél (a vektor elemeinek ismétlése a feltöltés során) .

7

Tömbök és mátrixok létrehozása (2)

- Példa:

```
x <- 1:6  
  
dim(x) <- c(2, 3) # egy két sorból és három oszlopból álló mátrix  
print(x)  
      [,1] [,2] [,3]  
[1,]    1    3    5  
[2,]    2    4    6  
  
dim(x) <- c(3, 2) # egy három sorból és két oszlopból álló mátrix  
print(x)  
      [,1] [,2]  
[1,]    1    4  
[2,]    2    5  
[3,]    3    6
```

6

Tömbök és mátrixok létrehozása (4)

- Tömbök létrehozásához használható az `array(x = NA, dim = length(x), dimnames = NULL)` függvény.
 - Az `x` vektor elemei alkotják a tömböt.
 - Ha `x` hossza kisebb a megadott méretnél, akkor az elemek ismétlése.
 - Üres bináris vektor esetén 0, az összes többi üres atomi vektor esetén megfelelő típusú NA értékekkel lesz feltöltve a tömb.
 - A tömbnek a vektor elemeivel feltöltése során először az első index futja be az összes lehetséges értéket, majd a második, ...
 - A `dim` és `dimnames` argumentumok az azonos nevű attribútumok értékét szolgáltatják.

8

Tömbök és mátrixok létrehozása (5)

- Mátrixok létrehozásához használható a `matrix(x = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)` függvény.
 - Az `x` vektor elemei alkotják a mátrixot.
 - Ha `x` hossza kisebb a megadott méretnél, akkor az elemek ismétlése (üres bináris vektor esetén 0, az összes többi üres atomi vektor esetén megfelelő típusú NA értékekkel lesz feltöltve a mátrix).
 - Az `nrow` és `ncol` argumentumok a sorok és oszlopok számát adják meg.
 - Elég az egyiket megadni, a másik meghatározható a vektor hosszából.
 - A vektor elemeivel a mátrix alapértelmezésben oszlop-folytonosan lesz feltöltve.
 - Ha a `byrow` argumentum értéke `TRUE`, akkor sor-folytonosan történik a feltöltés.

9

Függvények tömbök és mátrixok kezeléséhez

- Az `is.array(x)` és `is.matrix(x)` függvénnyel vizsgálható, hogy az adott objektum tömb-e illetve mátrix-e.
 - Az első függvény mátrixok esetén is `TRUE` értéket ad.
- Az `nrow(x)` és `ncol(x)` függvények az `x` objektum sorainak és oszlopainak számát adják.
 - Egy egyelemű egész vektor a visszatérési érték `dim` attribútummal rendelkező struktúrák (tömbök, mátrixok, adatkeretek) esetén, egyéb objektumok esetén pedig `NULL`.

11

Vektorok és egydimenziós tömbök

- Az atomi vektorok és az egydimenziós tömbök különböznek!
 - Az atomi vektoroknak nincs `dim`, sem `dimnames` attribútuma.
 - Példa:

```
x <- 1:10
y <- x
dim(y) <- c(1, 10)
all(x == y)      # TRUE, mivel x és y elemei megegyeznek
identical(x, y)  # FALSE, mivel x vektor, y pedig
                  # egy sorból álló mátrix
```

10

Függvények mátrixok kezeléséhez (1)

- A `diag(x)` függvény visszatérési értéke:
 - Ha `x` egyelemű egész vektor, akkor egy `x` sorból és oszlopból álló egységmátrix.
 - Egyéb egyelemű atomi vektorok egészszé konvertálása.
 - Ha `x` legalább kételemű vektor vagy egydimenziós tömb, akkor egy olyan diagonális mátrix, melynek főátlójában `x` elemei szerepelnek.
 - Ha `x` mátrix, akkor a főátlóban lévő elemekből képzett vektor.

12

Függvények mátrixok kezeléséhez (2)

- Mátrixok és vektorok egymáshoz illesztésére szolgálnak az `rbind(...)` és `cbind(...)` függvények.
 - Tetszőleges számú atomi vektort és mátrixot át lehet adni argumentumként.
 - Az első függvény „egymás alá” illeszti az argumentumként adott mátrixokat (vektorok sorvektorokként történő kezelése).
 - A második „egymás mellé” illeszti az argumentumként adott mátrixokat (vektorok oszlopvektorokként történő kezelése).
 - Az argumentumként adott mátrixok azonos számú sorból/oszlopból kell, hogy álljanak, a vektorok azonban lehetnek rövidebbek (kiegészítés az elemek ismétlésével).

13

Tömbök és mátrixok indexelése (1)

- Tömbök és mátrixok indexelése az `[]` operátorral hasonlóan történik a vektorokéhoz.
 - A korábban elmondottakat általánosítani lehet tömbökre és mátrixokra is.
- Különbségek vektorok indexeléséhez képest:
 - Az indexhatár-túllépés minden esetben hiba.
 - A `names` attribútum szerepét a `dimnames` attribútum veszi át (fejléc megjelenítésnél és karakter vektorokkal indexelésnél).

15

Függvények mátrixok kezeléséhez (3)

- A `t(x)` függvény transzponálást végez.
- Az `isSymmetric(x)` függvénnyel vizsgálható, hogy az adott mátrix szimmetrikus-e.
- A `det(x)` függvény az adott négyzetes numerikus mátrix determinánsát határozza meg.
- Az `eigen(x)` függvény az adott négyzetes numerikus vagy komplex mátrix sajátértékeit és sajátvektorait határozza meg.
- A `solve(a,b)` függvény az $a \%*\% x = b$ lineáris egyenletrendszert oldja meg.
 - x négyzetes numerikus vagy komplex mátrix, b négyzetes numerikus vagy komplex mátrix vagy vektor lehet.
 - `solve(x)` speciálisan az x mátrix invertálását végzi.

14

Tömbök és mátrixok indexelése (2)

- Az indexelés általában a dimenziókkal azonos számú index megadásával történik, `T[i, j, k, ...]` módon.
 - Bármelyik dimenzióban el lehet hagyni az indexet, mely az összes elem kiválasztását jelenti a megfelelő dimenzióban
 - Mátrixoknál az első index sorindex, a második oszlopindex.
- Egyetlen index megadása esetén vektorként történő indexelés.

16

Tömbök és mátrixok indexelése (3)

- Speciálisan meg lehet adni indexként egy olyan numerikus mátrixot, melynek a dimenziókkal megegyező számú oszlopa van.
 - Nem egész értékek egészé konvertálása.
 - Az eredmény egy vektor, az index mátrix valamennyi sora egy-egy kiválasztandó elem indexeit tartalmazza.
 - Az indexben nem megengedettek negatív értékek, 0 és NA viszont igen.
 - A nullákat tartalmazó sorok figyelmen kívül hagyása, NA értéket tartalmazó sorok NA értéket eredményeznek.

17

Példa mátrixok indexelésére (1)

- $m[1, 2]$:
 - Egyetlen elem kiválasztása.
- $m[1,]$:
 - Az első sor kiválasztása. Elemek kinyerésénél az eredmény megfelelő elemszámú vektor.
- $m[, 1]$:
 - Az első oszlop kiválasztása. Elemek kinyerésénél az eredmény megfelelő elemszámú vektor.
- $m[1:5,]$:
 - Az első öt sor kiválasztása. Elemek kinyerésénél az eredmény mátrix vagy megfelelő elemszámú vektor.

19

Tömbök és mátrixok indexelése (4)

- Elemek kinyerése esetén ha az indexelés eredményeként kapott struktúra mérete valamelyik dimenzióban 1, akkor a megfelelő dimenzió elhagyása.
 - Speciálisan ha az eredmény egydimenziós struktúra, akkor vektort kapunk.
 - Például mátrix egy sorát vagy oszlopát kiválasztva egy vektor az eredmény.

18

Példa mátrixok indexelésére (2)

- $m[1:2, 3:1]$:
 - Az első két sorban és első három oszlopban lévő elemek kiválasztása az oszlopok felcserélésével. Elemek kinyerésénél az eredmény mátrix.
- $m[abs(m) > 1]$:
 - Az egynél nagyobb abszolút értékű elemek kiválasztása. A mátrix vektorként indexelése történik a kifejezésben. Elemek kinyerésénél az eredmény vektor.

20

Infix és prefix operátorok (1)

- Az operátorok precedencia szerint csökkenő sorrendben:
 - ::
 - \$ @
 - ^
 - + - (egy operandusú)
 - :
 - %xyz%
 - * /
 - + - (kétooperandusú)

21

Infix és prefix operátorok (3)

- A ^, (egy operandusú) -, <-, = és <<- operátor jobbról, az összes többi balról asszociatív.
- Precedencia előírására a programozási nyelvekben megszokott módon lehet használni a kerek zárójeleket.
- A felhasználó is definiálhat infix kétooperandusú operátorokat.
 - Az ilyen operátorok %xyz% alakúak, ahol xyz tetszőleges, a '%' karaktertől különböző nyomatható karakterekből álló karaktersorozat.

23

Infix és prefix operátorok (2)

- Az operátorok precedencia szerint csökkenő sorrendben (folytatás):
 - > >= <= < == !=
 - !
 - & &&
 - | ||
 - ~ (egy és kétooperandusú)
 - -> ->>
 - =
 - <- <<-

22

Aritmetikai operátorok (1)

- Kétooperandusú műveleteknél az eredmény:
 - Komplex, ha az egyik vagy mindkét operandus komplex típusú.
 - Valós, ha az egyik vagy valamelyik operandus valós típusú.
 - Egész, ha mindkét operandus egész típusú, a ^ és / operátorok kivételével, melyek minden esetben valós eredményt adnak.
- Az operandusok lehetnek logikai típusúak, melyek automatikusan egész típusúvá lesznek konvertálva.
 - TRUE érték 1 lesz, FALSE pedig 0.

24

Aritmetikai operátorok (2)

- Az aritmetikai operátorok vektorokra alkalmazása elemenként alkalmazást jelent.
 - Az operandusok azonos hosszra normalizálása.

25

Aritmetikai operátorok (4)

- A `%**` operátor szolgál mátrixok szorzására.
 - Hiba, ha a mátrixok mérete nem megfelelő.
 - Ha az egyik operandus vektor, akkor annak egy sorból vagy oszlopból álló mátrixszá alakítása, hogy a művelet elvégezhető legyen.
 - Ha mindkét operandus vektor, akkor azok belső szorzatát adja (az eredmény egy egy sorból és oszlopból álló mátrix).

27

Aritmetikai operátorok (3)

- A `+`, `-`, `*`, `/` operátorok jelentése a programozási nyelvekben megszokott.
 - Egészek osztásának eredménye valós.
- A `^` operátor hatványozást jelent.
- A `%%` operátor maradékos osztást jelent.
 - 0-val való maradékos osztás eredménye NaN (ha mindkét operandus egész, akkor NA).
- A `%/%` operátor egész osztást jelent.
 - 0-val való egész osztás eredménye Inf (ha mindkét operandus egész, akkor NA).

26

Logikai operátorok (1)

- A logikai operátorok automatikusan logikai típusúvá konvertálják az operandusokat.
 - Komplex, valós és egész típusú értékek konvertálásakor minden nullától különböző érték TRUE, a 0 pedig FALSE értéket eredményez.
 - Kivétel a NaN érték, mely speciálisan NA értéket eredményez.
 - Bináris vektorok esetében a `!`, `&` és `|` operátoroknál nincs konverzió, bitenkénti műveletvégzés történik.
 - Karakter vektorok esetén hiba.
- Operandusként megjelenhet NA.

28

Logikai operátorok (2)

- A `!` operátor elemenkénti negációt végez.
- Az `&` és `|` operátorok elemenkénti logikai ÉS illetve logikai VAGY műveletet végeznek.
- Az `&&` és `||` operátorok ugyancsak logikai ÉS illetve VAGY műveletek.
 - Azonban kizárólag az operandusok első elemeit használják és rövidzár kiértékelést végeznek.
 - Használat vezérlési szerkezetekben (`if`, `while`).
- Az `xor(x, y)` függvény az elemenkénti kizáró VAGY logikai műveletet valósítja meg.

29

Néhány hasznos logikai függvény (2)

- Az `ifelse(feltétel, igaz, hamis)` függvény egy *feltétel*-lel megegyező alakú értéket ad, melynek elemei az *igaz* és *hamis* argumentumokból lesznek kiválasztva *feltétel* alapján.
 - Az első argumentum logikai típusú vagy ilyen típusúvá konvertálható kell, hogy legyen.
 - Az eredmény hossza és attribútumai megegyeznek *feltétel*-ével (ha például *feltétel* mátrix, akkor az eredmény is az).
 - Szükség esetén *igaz* és *hamis* hosszának normalizálása *feltétel* hosszára.
 - Ha *feltétel* minden eleme hamis, akkor az *igaz* argumentum nem lesz kiértékelve, hasonlóan ha *feltétel* minden eleme igaz, akkor *hamis* nem lesz kiértékelve.
 - Ha *feltétel*-ben NA szerepel, az eredmény megfelelő eleme is NA lesz.

31

Néhány hasznos logikai függvény (1)

- Az `all(...)` függvény akkor, és csak akkor ad TRUE értéket, ha az argumentumként adott logikai vektorok minden eleme TRUE értékű.
 - Példa: `all(x > 0)`, `all(y > 0, z > 0)`
- Az `any(...)` függvény akkor, és csak akkor ad TRUE értéket, ha az argumentumként adott logikai vektorok elemei között van legalább egy TRUE értékű.
 - Példa: `any(x != 0)`
- Mindkét függvény logikai típusúvá alakítja az argumentumait.

30

Néhány hasznos logikai függvény (3)

- Példa:

```
age <- c(11, 73, 38, 63, 13)
ifelse(age >= 18, "adult", "underage")
# "underage" "adult" "adult" "adult" "underage"

x <- runif(100, -1, 1)
x <- ifelse(abs(x) > 0.5, x, 0)
all(x == 0 | 0.5 < abs(x)) # TRUE

ifelse(-1 <= y & y <= 1,
      -abs(y) + 1,
      abs(y) - 1
    )
```

32

Összehasonlító operátorok (1)

- A `>`, `>=`, `<=`, `<`, `==`, `!=` operátorok jelentése a programozási nyelvekben megszokott.
 - Karakterláncok hasonlítása lexikografikus rendezés szerint (a rendezés módja a környezettől függ).
- Az operandusok elemenkénti összehasonlítása.
- Legalább az egyik operandus atomi vektor kell, hogy legyen.
 - Ha a két operandus különböző típusú atomi vektor, akkor a művelet elvégzéséhez azonos típusúvá lesznek konvertálva (a céltípus a konverzió során a karakter, komplex, valós, egész és logikai típusok közül sorrendben az első).
 - Ha az egyik operandus lista, akkor annak az atomi vektor típusára konvertálása (hiba, ha ez nem végezhető el).
 - A konverzió elvégezhető, ha a lista valamennyi eleme egyelemű és a megfelelő típusúvá konvertálható.

33

Értékadó operátorok (1)

- A `név <- érték`, `érték -> név`, `név <<- érték` és `érték ->> név` kifejezések egyaránt az adott nevű változónak adnak értéket.
 - `név` egy változó neve, `érték` egy tetszőleges kifejezés.
- A `<-` és `<<-` operátorok jobbról, a `->` és `->>` operátorok pedig balról asszociatívak.
- Minden esetben az *érték* kifejezés értéke az értékadó kifejezések értéke.
 - Tehát megengedettek például a `z <- y <- x <- 1` és `1 -> x -> y -> z` kifejezések

35

Összehasonlító operátorok (2)

- NA értékkel összehasonlítás eredménye minden esetben NA!
- Az `if` és `while` utasításokban – melyekben a feltétel egyetlen logikai érték kell, hogy legyen – kerülni ajánlott az `==` és `!=` operátorok használatát, melyek végrehajtása elemenként történik.
 - Használjuk helyettük az `identical(x, y)` függvényt, mely TRUE értéket ad vissza, ha a két argumentum azonos, egyébként pedig FALSE értéket.
 - Gyakran még hasznosabb a megengedőbb `isTRUE(all.equal(x, y))`.

34

Értékadó operátorok (2)

- A `<-` és `->` operátorok esetében az értékadás abban a környezetben történik, melyben az értékadó kifejezés kiértékelésre kerül.
- A `<<-` és `->>` operátorok az adott nevű szimbólumot keresik abból a környezetből kiindulva, melyben végrehajtásra került az értékadás, kifelé haladva a bezáró környezetekben.
 - Ha valamelyik környezetben van ilyen szimbólum, akkor az értékének helyettesítése az értékadásban adott kifejezés értékével.
 - Ha nincs ilyen szimbólum, akkor az értékadás elvégzése globálisan, a globális környezetben.

36

Értékadó operátorok (3)

- Az $x \leftarrow \text{érték}, \text{érték} \rightarrow x, x \leftarrow \leftarrow \text{érték}$ és $\text{érték} \rightarrow x$ kifejezésekben x lehet egy objektum egy részét megadó kifejezés is.
 - Megengedettek például az $x[1:10] \leftarrow 0$ és $x[10:19] \leftarrow 1:10$ értékadások.

37

Környezetek (2)

- A környezetek kezelése automatikusan történik.
 - Például minden függvényhívás során automatikusan létrejön egy környezet, mely a függvény lokális változóit és argumentumait tartalmazza.
- Vannak azonban környezetek kezelésére szolgáló függvények.
 - Például a `new.env()` függvény egy új, üres környezetet hoz létre.

39

Környezetek (1)

- A környezeteket egy szimbólum-érték párokat tartalmazó úgynevezett keret (*frame*) alkotja, valamint egy mutató egy másik környezetre, amit bezáró környezetnek hívnak (*enclosure*).
- Ilyen módon a környezetek faszerkezetet alkotnak.
 - A fa gyökere egy üres környezet, mely az `emptyenv()` függvénnyel érhető el.
 - A fában egy környezet szülője a bezáró környezete.
- Szimbólumkeresés művelet (szimbólum értékének meghatározása).
 - Ha egy környezet nem tartalmazza a szimbólumot, a keresés a bezáró környezetben folytatódik.

38

Környezetek (3)

- Az úgynevezett globális környezet a felhasználó munkaterülete.
 - Minden parancssorban elvégzett értékadás a globális környezetben hozza létre a megfelelő objektumot.
 - Ez a keresési útvonal első eleme, a keresési útvonalban ezt követő környezet a bezáró környezete, és így tovább (lásd a keresési útvonalnál leírtakat).
- A `.GlobalEnv` változó és `globalenv()` függvény értéke egyaránt a globális környezetet szolgáltatják.

40

Környezetek (4)

- Példa környezetek használatára:

```
ls()
ls(pattern = "x")
ls(pattern = "^x")
rm(list = ls(all = TRUE))
exists("x")      # FALSE
x <- 1
exists("x")      # TRUE
exists("sin")    # TRUE
exists("sin", inherits = FALSE) # FALSE
if (! exists("f", mode = "function"))
  f <- function(x) 1 / (1 + exp(-x))
# ...
rm(f)
```

41

Függvények

- Minden függvényt az alábbi három komponens alkot:
 - Formális argumentumlista
 - Törzs
 - Egy a függvény környezetének nevezett környezet
- Egy függvényt és környezetét együtt lezárásnak (*closure*) nevezik.
- Függvények komponenseinek manipulálására szolgálnak a `formals(f)`, `body(f)` és `environment(f)` függvények.
 - Ezeket akár értékadó kifejezésben is lehet használni értékadó operátor bal oldalán.

43

Keresési útvonal

- Környezeteket tartalmaz, melyekben a máshol nem megtalált szimbólumokat kell keresni.
- A keresési útvonal első eleme a globális környezet, utolsó eleme pedig mindig a base csomag.
- A keresési útvonalon szereplő valamennyi környezetnek az útvonalon őt követő környezet a bezáró környezete.
- Ha egy szimbólumkeresés sikertelen volt, akkor a keresési útvonalban adott környezetekben folytatódik a keresés.
- A `search()` függvény a keresési útvonalat adja vissza egy karakter vektor formájába.
- A keresési útvonalat manipulálják például az `attach()`, `detach()` és `library()` függvények.

42

Függvények definiálása (1)

- Egy névtelen függvényt definiál a `function(arglista) törzs` kifejezés.

44

Formális argumentumlista

- Olyan vessző karakterekkel elválasztott argumentumok alkotják, melyek az alábbiak lehetnek:
 - *név*
 - *név* = kifejezés
 - ...
- A második alapértelmezett értéket ad az argumentumnak.
- A harmadik változó argumentumszámú függvényeknél és argumentumlista továbbadásnál használt.
- Az argumentumlista lehet üres.

45

Függvények definiálása (2)

- Példa:

```
p <- function(a, x) sum(x^((length(a) - 1) : 0) * a)

fibonacci <- function(n) {
  a <- (1 + sqrt(5)) / 2
  b <- (1 - sqrt(5)) / 2
  (a^n - b^n) / sqrt(5)
}

d <- function(x, y) {
  if (! is.vector(x) || ! is.vector(y)
      || ! is.numeric(x) || ! is.numeric(y))
    stop("both arguments must be numeric vectors")
  z <- sqrt((x - y) %*% (x - y))
  drop(z)
}
```

47

Függvények törzse

- A törzs a függvény visszatérési értékét szolgáltatja, lehet egyetlen kifejezés vagy összetett utasítás.
- Kizárólag függvény törzsében használható a `return(x)` függvényhívás.
 - A függvény befejeztetésére szolgál, az argumentum értéke lesz a visszatérési érték.
 - A függvényt meg lehet hívni argumentum nélkül, `return()` módon is, ekkor `NULL` a visszatérési érték.
- Ha a függvény törzsében nem hajtódik végre egyetlen `return()` függvényhívás sem, akkor a visszatérési érték a törzsben utoljára kiértékelt kifejezés értéke.

46

Függvények definiálása (3)

- Példa:

```
f <- function(beta = 1)
  function(x) 1 / (1 + exp(-beta * x))
```

48

Függvényhívás (1)

- A következő formában lehetséges:
függvény-hivatkozás(*arg*₁, ..., *arg*_{*n*}).
- A kifejezésben szereplő *függvény-hivatkozás* legegyszerűbb esetben egy azonosító (a függvény neve), de lehet karakterlánc (ha a függvény neve nem azonosító) vagy egy függvény objektumot szolgáltató kifejezés.
- Az argumentumokat meg lehet adni névvel is *név* = *kifejezés* módon, ekkor a paraméterkiértékelésnél név szerinti kötés történik.
- Legegyszerűbb esetben *név* azonosító, de lehet karakterlánc is.
- Továbbá argumentum helye hagyható üresen és megadható a speciális ... karaktersorozat.
 - Utóbbi csak akkor megengedett, ha a hívás függvény törzsében történik.

49

Függvényhívás (3)

- Példa:

```
f <- function(beta=1)
  function(x) 1 / (1 + exp(-beta * x))
f()
f()(0)
f(1)
f(1)(0)

"Hello, world!" <- function() cat("Hello, world!\n")
get("Hello, world!")
"Hello, world!"()

(function(n, k) factorial(n) / (factorial(k) *
  factorial(n - k))) (9, 3)
```

51

Függvényhívás (2)

- Bizonyos speciális függvényhívások értékadó operátor bal oldalán is megjelenhetnek értékadó kifejezésben.
 - Lásd például az attribútumok értékének beállítása kapcsán leírtakat.
- Lásd a kiértékelési környezeténél leírtakat a függvények és környezetek kapcsán.

50

Rekurzió (1)

- Rekurzió megvalósításához használjuk a `Recall(...)` függvényt, mely a tartalmazó függvényt fogja meghívni.
 - Egyébként a függvény átnevezése hibát okoz.

52

Rekurzió (2)

- Rekurzió hibás megvalósítása (a függvény nem átnevezhető):

```
h <- function(n) {  
  if (n == 1)  
    return(1)  
  else  
    return(2 * h(n - 1) + 1)  
}  
  
h(3)  
hanoi <- h    # A függvény lemásolása  
rm(h)         # A h függvény törlése  
Hanoi(3)      # Hiba, mivel nincs h függvény!
```

53

Rekurzió (4)

- Rekurzió helyes megvalósítása (Ackermann-függvény):

– Lásd:

https://en.wikipedia.org/wiki/Ackermann_function

```
ackermann <- function(m, n) {  
  if (m == 0)  
    return(n + 1)  
  else if (m > 0 && n == 0)  
    return(Recall(m - 1, 1))  
  else if (m > 0 && n > 0)  
    return(Recall(m - 1, Recall(m, n - 1)))  
  stop("Should never get here")  
}
```

55

Rekurzió (3)

- Rekurzió helyes megvalósítása:

```
f <- function(n) {  
  if (n == 1)  
    return(1)  
  else  
    return(2 * Recall(n - 1) + 1)  
}  
  
f(3)  
hanoi <- f    # A függvény átnevezése  
rm(f)         # Az f függvény törlése  
hanoi(3)
```

54

Függvények és környezetek (1)

- Egy függvény környezete az a környezet, mely a függvény létrehozásakor aktív volt.
 - A függvény környezetének szimbólumaihoz a függvény létrehozásának pillanatában aktuális értékeik kerülnek rögzítésre.
- A parancssorban létrehozott függvények környezete ilyen módon a globális környezet.
- Az `environment(f)` függvényhívás az argumentumként adott függvény környezetét szolgáltatja.

56

Függvények és környezetek (2)

- Minden függvényhívás során automatikusan létrejön egy kiértékelési környezetnek nevezett új környezet, amely a függvény lokális változóit és argumentumait tartalmazza.
 - Ebben a környezetben történik a függvény törzsét alkotó kifejezések kiértékelése.
 - Ennek a környezetnek a bezáró környezete a függvény környezete.

57

Paraméterkiértékelés (1)

- Az aktuális paraméterek formális paraméterekhez való hozzárendelése név szerinti és sorrendi kötés alapján történik.
- Mivel formális argumentumokhoz meg lehet adni alapértelmezett értéket, a hívásnál megadott argumentumok száma lehet kevesebb a formális argumentumok számánál.
- Függvény lehet változó argumentumszámú is.

59

Függvények és környezetek (3)

- Példa:

```
rm(list = ls())

f <- function(x) {
  g <- function(y) x + y
  return(g)
}

h <- f(3)
ls(envir = environment(f))      # "f" "h"
ls(envir = environment(h))      # "g" "x"
get("x", envir = environment(h)) # 3
```

58

Paraméterkiértékelés (2)

- Az aktuális és formális argumentumok megfeleltetése három lépésben történik az alábbiak szerint:
 1. Teljes névegyezés alapján
 2. Részleges névegyezés alapján
 3. Sorrendi kötés alapján
- Hiba, ha az algoritmus végén nem lesz párja egy aktuális argumentumnak, vagy egy olyan formális paraméternek, melynek nincs alapértelmezett értéke.

60

Paraméterkiértékelés (3)

- Teljes névegyezés alapján:
 - Minden névvel megadott aktuális argumentumot meg kell feleltetni az azonos nevű formális argumentumnak.
 - Hiba, ha egy formális argumentum nevével több aktuális argumentum neve is megegyezik.
 - Azaz minden formális argumentum értéke legfeljebb egyszer adható meg.

61

Paraméterkiértékelés (5)

- Sorrendi kötés alapján:
 - Az előző két lépésben kimaradt formális argumentumok megfeleltetése a megmaradt aktuális argumentumoknak a megadásuk sorrendjében.
 - Ha van a formális argumentumlistában . . . , akkor ennek megfeleltetni az összes kimaradt aktuális argumentumot.

63

Paraméterkiértékelés (4)

- Részleges névegyezés alapján:
 - Az előző lépésben formális argumentumoknak nem megfeleltetett névvel megadott aktuális argumentumok mindegyikéhez olyan formális argumentumot kell keresni a megmaradtak közül, melynek neve az adott aktuális argumentum nevével kezdődik.
 - Hiba, ha egy aktuális argumentumhoz több ilyen formális argumentum van.
 - Tehát név szerinti paraméterátadásnál nem kötelező kiírni egy formális argumentum teljes nevét, ha az egyértelműen rövidíthető.
 - Ha a formális argumentumok között van . . . , akkor a részleges névegyezés vizsgálata csak az ezt megelőző formális argumentumoknál.

62

Paraméterkiértékelés (6)

- Az aktuális argumentumok és az alapértelmezett argumentumok kiértékelése másként történik.
 - Az aktuális argumentumok kiértékelése a hívó függvény környezetében történik.
 - Az alapértelmezett argumentumok kiértékelése pedig a függvény környezetében.
- Egy argumentum kiértékelése csak akkor történik meg, ha szükség van az értékére.
 - Elképzelhető, hogy egyáltalán nem is értékelődik ki egy aktuális argumentum.
 - Ezért ne adjunk aktuális paraméterként olyan mellékhatásos kifejezést, mint például az értékadás.

64

Paraméterkiértékelés (7)

- Az aktuális és formális argumentumok megfeleltetése egymásnak nem a leírtak szerint történik a primitív függvények esetében.
 - Ezek tipikusan nem veszik figyelembe a névvel megadott aktuális argumentumok neveit, hanem sorrendi kötés alapján feleltetik meg egymásnak a formális és aktuális argumentumokat.
- A paraméterátadás érték szerint történik.

65

Függvények vektorizálása (2)

- Példa: a korábban definiált függvények vektorizálása

```
Vectorize(hanoi)(1:10)
# 1 3 7 15 31 63 127 255 511 1023

Vectorize(ackermann)(0:3, 0)
# 1 2 3 5

outer(0:3, 0:3, Vectorize(ackermann))
#      [,1] [,2] [,3] [,4]
# [1,] 1    2    3    4
# [2,] 2    3    4    5
# [3,] 3    5    7    9
# [4,] 5   13   29   61
```

67

Függvények vektorizálása (1)

- Számos operátor és függvény vektorokra alkalmazása elemenkénti végrehajtást eredményez.
 - Ilyenek például az aritmetikai operátorok és a matematikai függvények egy része (`abs(x)`, `sin(x)`, `sqrt(x)`, ...).
- A `Vectorize(f)` függvénnyel lehet vektorizálni az argumentumként adott `f` függvényt.
 - A visszatérési érték egy az `f` függvénnyel egyező formális argumentumlistájú függvény, mely azonban vektorokra is alkalmazható.
 - Vektorokra való alkalmazás `f` elemenkénti alkalmazását jelenti.

66

Összetett utasítás

- Blokknak vagy összetett utasításnak nevezik az alábbi:
`{ kifejezés1 ; kifejezés2 ; ... ; kifejezésn }`
- Az összetett utasításban tetszőleges kifejezéseket meg lehet adni, melyeket el lehet választani új sor karakterekkel is.
- A kifejezések kiértékelése, az összetett utasítás értéke a benne utoljára kiértékelt kifejezés értéke.
- Az összetett utasítás kiértékelése csak akkor történik meg, ha a végét jelző `'}'` karakter után beolvasásra került egy új sor karakter.

68

Vezérlési szerkezetek

- Elágaztató utasítások:
 - `if ( kifejezés1 ) kifejezés2 else kifejezés3`
 - `if ( kifejezés1 ) kifejezés2`
- Ciklusszervező utasítások:
 - `repeat kifejezés`
 - `while ( kifejezés1 ) kifejezés2`
 - `for ( név in kifejezés1 ) kifejezés2`
 - `break`
 - `next`

69

Az if utasítás (2)

- Végrehajtáskor először kiértékelődik az első kifejezés.
 - Ha `kifejezés1` értéke olyan logikai vektor, melynek első eleme TRUE, akkor kiértékelődik `kifejezés2` és annak értéke az utasítás visszatérési értéke.
 - Egyébként:
 - Ha van `else` ág, akkor kiértékelődik `kifejezés3` és annak értéke az utasítás visszatérési értéke.
 - Ha nincs `else` ág, akkor NULL az utasítás visszatérési értéke.

71

Az if utasítás (1)

- Használat:
 - `if ( kifejezés1 ) kifejezés2 else kifejezés3`
`if ( kifejezés1 ) kifejezés2`
 - `kifejezés1` értéke olyan logikai vagy azzá konvertálható vektor lehet, melynek első eleme NA értéktől különböző kell, hogy legyen.
 - `kifejezés2` és `kifejezés3` tetszőleges kifejezések.
 - Figyelmeztetést eredményez, ha `kifejezés1` értéke egynél hosszabb vektor (csak az első elem lesz felhasználva).

70

Az if utasítás (3)

- Ha az `if` utasítás nem összetett utasításban szerepel, akkor az `else` egy sorban kell, hogy szerepeljen `kifejezés2`-vel.
 - Speciálisan ha `kifejezés2` összetett utasítás, akkor a végét jelző `' } '` és az `else` között nem megengedett új sor karakter.
- Az `if` utasításokat tetszőleges mélységig egymásba lehet ágyazni.

72

Az if utasítás (4)

- Példa:

```
med <- function(x) {  
  x <- sort(x)  
  if (length(x) %% 2 == 1)  
    x[ceiling(length(x) / 2)]  
  else  
    (x[length(x) / 2] + x[length(x) / 2 + 1]) / 2  
}  
  
med(1:10) # 5.5
```

73

A repeat utasítás (2)

- Példa:

```
env <- new.env()  
repeat {  
  line <- readLines(n = 1)  
  line <- trimws(line)  
  if (line == "")  
    next  
  try(print(eval(parse(text = line), envir = env)))  
}
```

75

A repeat utasítás (1)

- Használat:

repeat *kifejezés*

- Az utasításban szereplő kifejezés tetszőleges lehet.
- A kifejezés kiértékelése újra és újra, melyet egy break utasítás végrehajtása szakít meg.
- A kifejezés tipikusan egy feltételes break utasítást is tartalmazó összetett utasítás (lehet önmagában egyetlen break utasítás, de ez nem túl hasznos).
- Végtelen ciklus megvalósítására szolgál.
- Az utasítás visszatérési értéke NULL.

74

A while utasítás (1)

- Használat:

while (*kifejezés₁*) *kifejezés₂*

- Az utasításban szereplő *kifejezés₁* kifejezésre pontosan ugyanazok a feltételek kell, hogy teljesüljenek, mint az if utasítás esetében.
- *kifejezés₂* hasonlóan tetszőleges kifejezés lehet.
- A programozási nyelvekben megszokott kezdőfeltételes ciklus.

76

A while utasítás (2)

- Végrehajtáskor először kiértékelődik az első kifejezés.
 - Ha *kifejezés₁* értéke olyan logikai vektor, melynek első eleme TRUE, akkor kiértékelődik *kifejezés₂*, majd újból *kifejezés₁*, ...
 - Az utasítás végrehajtása akkor fejeződik be, ha *kifejezés₁* kiértékelése FALSE értéket eredményez.
- Az utasítás visszatérési értéke NULL.

77

A for utasítás (1)

- Használat:
`for ( név in kifejezés1 ) kifejezés2`
 - *kifejezés₁* olyan kifejezés lehet, melynek értéke atomi vektor, lista, kifejezés objektum vagy NULL.
 - *kifejezés₂* tetszőleges kifejezés lehet.

79

A while utasítás (3)

- Példa:

```
gcd <- function(m, n) {  
  r <- m %% n  
  while (r != 0) {  
    m <- n  
    n <- r  
    r <- m %% n  
  }  
  n  
}  
  
gcd(51877827, 26314496) # 7907
```

78

A for utasítás (2)

- Végrehajtáskor az adott nevű változó sorban felveszi értékül *kifejezés₁* értékének elemeit és minden elemre kiértékelődik *kifejezés₂*.
 - Csak egyszer kerül kiértékelésre *kifejezés₁*!
 - Ha *kifejezés₁* értéke NULL, akkor egyszer sem értékelődik ki *kifejezés₂*.
 - *kifejezés₂*-ben meg lehet változtatni a változó értékét, de az ettől függetlenül felveszi *kifejezés₁* értékének minden elemét.
- Az utasítás visszatérési értéke NULL.

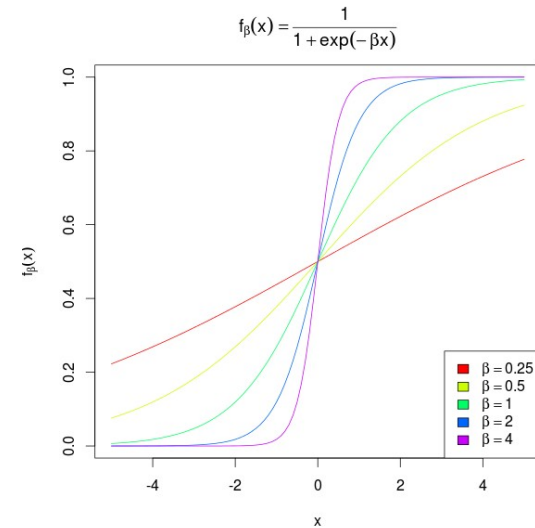
80

A for utasítás (3)

- Az utasítás végrehajtása után mellékhataásként a változó értéke az lesz, melyet a kiértékelés során utoljára felvett.
 - Speciálisan a változó értéke NULL lesz akkor, ha *kifejezés₁* értéke 0 hosszú vagy pedig NULL.

81

A for utasítás (5)



83

A for utasítás (4)

- Példa:

```
f <- function(beta) function(x) 1 / (1 + exp(-beta * x))
plot(f(1), -5, 5,
     type = "n",
     ylab = expression(f[beta](x)))
title(main = expression(f[beta](x) ==
  frac(1, 1 + exp(-beta * x))))
beta <- 2^(-2:2)
palette <- rainbow(length(beta))
for (i in 1:length(beta)) {
  plot(f(beta[i]), -5, 5, col = palette[i], add = TRUE)
}
legend <- as.expression(lapply(beta,
  function(x) substitute(beta == value, list(value = x))))
legend("bottomright", legend = legend, fill = palette)
```

82

A break és next utasítások

- Mindkét utasítás csak ciklusszervező utasításokban használható.
- A break utasítás megszakítja a legbelső tartalmazó ciklusszervező utasítás végrehajtását.
- A next utasítás megszakítja a ciklus aktuális iterációjának végrehajtását és a végrehajtás következő iterációját idézi elő.
- Nincs visszatérési értékük.

84