Az R programozási nyelv (tárgyalt R verzió: 2.10.0)

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

Azonosítók

- Legalább egy hosszúságú karaktersorozatok, amelyek betű, számjegy, pont és aláhúzójel karakterekből állnak
- A környezettől függ, hogy milyen karakterek számítanak betűknek
- Nem kezdődhetnek számjegy vagy aláhúzójel karakterrel
- Ha pont karakterrel kezdődnek, akkor a második karakter nem lehet számjegy
- A pont karakterrel kezdődő azonosítókat alapértelmezésben nem listázza az `ls()` függvény
- Kisbetűk és nagybetűk megkülönböztetése

Kulcsszavak

- Az alábbi azonosítókat tilos objektumok neveként használni:
  - `if`, `else`, `repeat`, `while`, `function`, `for`, `in`, `next`, `break`, `TRUE`, `FALSE`, `NULL`, `Inf`, `NaN`, `NA`, `NA_integer_`, `NA_real_`, `NA_complex_`, `NA_character_`
  - `...`, `..1`, `..2`, `..3`, …

Objektumok

- Az R által kezelt különböző memóriában tárolt adatszerkezeteket objektumoknak nevezzük
- Lehetnek például vektorok, listák, kifejezés objektumok, környezetek, …
  - Ezeken túl még számos további fajta objektum van, amelyek egy részével az átlagfelhasználó egyáltalán nem is találkozik

Objektumok típusa, konverziók (1)

- A `typeof(`*x*`)` függvény visszaadja az argumentumként kapott objektum típusát
  - A visszatérési érték egy karakterlánc, például `"character"`
- Logikai visszatérési értékű függvények, amelyekkel vizsgálható, hogy az argumentum a megfelelő típusú-e:
  - `is.logical(`*x*`)`, `is.integer(`*x*`)`, `is.double(`*x*`)`, `is.complex(`*x*`)`, `is.character(`*x*`)`, `is.numeric(`*x*`)`, `is.list(`*x*`)`, `is.function(`*x*`)`, `is.symbol(`*x*`)`, ...

Objektumok típusa, konverziók (2)

- A típusok között elég szabad átjárás van
  - Kifejezések kiértékelése során automatikus típuskonverziók
  - Típuskonverziók kényszerítésére számos konverziós függvény áll rendelkezésre
- Konverziós függvények:
  - `as.logical(`*x*`)`, `as.integer(`*x*`)`, `as.double(`*x*`)`, `as.complex(`*x*`)`, `as.character(`*x*`)`, `as.raw(`*x*`)`, `as.numeric(`*x*`)`, `as.list(`*x*`)`, `as.function(`*x*`)`, `as.symbol(`*x*`)`, ...

Vektorok (1)

- A legalapvetőbb fajta objektumok a vektorok
  - A tárban folytonosan elhelyezkedő elemekből állnak, amelyekhez indexeléssel lehet hozzáférni
  - Nincsenek a vektoroknál egyszerűbb objektumok, a konstansok egyelemű vektorokat reprezentálnak

Vektorok (2)

- Legegyszerűbb esetben a vektorok azonos típusú elemekből állnak, ezek az úgynevezett atomi vektorok, amelyeknek az alábbi 6 fajtája van:
  - logikai (`"logical"`)
  - egész (`"integer"`)
  - valós (`"double"`)
  - komplex (`"complex"`)
  - karakter (`"character"`)
  - bináris (`"raw"`)
- Zárójelben a `typeof()` függvény által adott típus

Vektorok (3)

- Az egész és valós vektorokat közös néven numerikus vektoroknak hívjuk
- Karakter vektorok elemeit karakterláncoknak nevezzük
- Vektorokat a `c(…)` függvénnyel lehet létrehozni
  - Közös típusra konvertálja és egyetlen vektorrá fűzi össze az argumentumokat
  - Például:

```
c("X","Y","Z")              "X" "Y" "Z"
c(c(1,2),3,c(4,5))          1 2 3 4 5
c(0i,TRUE,2)                0+0i 1+0i 2+0i
c(6,"Days",7,"Nights")      "6" "Days" "7" "Nights"
```

Vektorok (4)

- Vektorokat hoznak létre az `integer(`*n*`)`, `double(`*n*`)`, `complex(`*n*`)` és `raw(`*n*`)` függvények
  - Az eredmény egy olyan megfelelő típusú *n*-elemű vektor, amelynek minden eleme `0`
- A `logical(`*n*`)` függvény egy *n*-elemű logikai vektort ad, amelynek minden eleme `FALSE`
- A `character(`*n*`)` függvény egy *n*-elemű karakter vektort ad, amelynek minden eleme üres karakterlánc
- Ha az *n* argumentum értéke `0`, akkor az eredmény egy megfelelő típusú üres vektor
  - Tehát több különböző típusú üres vektor van

Vektor-aritmetika (1)

- Vektorok bárhol előfordulhatnak aritmetikai kifejezésekben, ekkor a műveletek elemenként lesznek végrehajtva
  - Van néhány speciális művelet, amely kivételt képez, például %*% vektorok esetén a belső szorzatot adja (az eredmény egy egy sorból és oszlopból álló mátrix)
- Az aritmetikai függvények vektorokra alkalmazása is elemenkénti alkalmazást jelent
- Nagyon komplex kifejezéseket nagyon tömören le lehet írni ilyen módon

Vektor-aritmetika (2)

- Kifejezésekben szereplő vektorok normalizálása közös hosszra
  - Ez a rövidebb vektor elemeinek ismétlését jelenti: addig ismétlődnek az elemek, amíg el nem érjük a hosszabb vektor hosszát
  - Ha a rövidebb vektor elemeinek száma nem osztja a hosszabb vektor elemeinek a számát, akkor is elvégzésre kerül a művelet, de figyelmeztetést kapunk
  - Speciális eset az, amikor a rövidebb vektor üres vektor, ekkor az eredmény is üres vektor

Vektor-aritmetika (3)

- Példák:

```
c(1,2,3,4)*c(2,3,4,5)    2 6 12 20

c(5,6,7,8)*2             10 12 14 16

2^(0:7)                  1 2 4 8 16 32 64 128

cos(c(-pi,0,pi))         -1 1 -1

sin(seq(from=0,to=pi/2,length=100))

exp(-1)/factorial(0:5)

c(1,2,3)%*%c(1,2,3)      14
```

Vektorok indexelése

- Az x vektor indexelése *x*[*i*] vagy *x*[[*i*]] módon történhet, ahol az *i* index egész, valós, logikai vagy karakter típusú lehet
  - Bizonyos esetben az index elhagyható vagy megadható `NULL`
  - Az első elem kiválasztásához az 1 indexet kell megadni
  - Valós indexek egésszé konvertálása
- Az indexelés nem csak az elemek értékének kinyerésére, hanem az elemek helyettesítésére is használható

Vektorok indexelése `[[]]` operátorral

- `[[]]` csak egyetlen elem kiválasztására használható
  - Indexként csak egyelemű vektor adható meg
  - Logikai típusú index egésszé konvertálása
    - Azaz *x*`[[TRUE]]` ugyanaz, mint *x*`[[1]]`, *x*`[[FALSE]]` pedig mint *x*`[[0]]` (utóbbi hibát eredményez)
  - Ennél az operátornál az indexhatár-túllépés nem megengedett

Vektorok indexelése `[]` operátorral (1)

- A `[]` operátorral egyidejűleg több elemet is ki lehet választani
- Az index tetszőleges elemszámú vektor lehet
- Az indexet akár el is lehet hagyni, ami az összes elem kiválasztását jelenti
- Speciálisan megjelenhet indexként `NULL`
  - Ez ekvivalens azzal az esettel, amikor az index az `integer(0)` kifejezés által adott üres egész vektor
  - Az eredmény megfelelő típusú üres vektor
- Az indexhatár-túllépés megengedett, megfelelően kezelt

Vektorok indexelése `[]` operátorral (2)

- Vektor elemeinek kinyerésekor a `names`, `dim` és `dimnames` attribútumok kivételével elhagyásra kerül valamennyi attribútum
  - Kivéve azt az eset, amikor az index nincs megadva

Vektorok indexelése [] operátorral (3)

- Ha az index pozitív egészekből álló vektor, akkor a megfelelő elemek kiválasztása az adott sorrendben
  - A vektor hosszánál nagyobb index NA értéket eredményez
- Ha az index negatív egészekből álló vektor, akkor azokat az elemeket adja meg, amelyek nem lesznek kiválasztva, az összes többi igen
  - A vektor hosszánál nagyobb abszolút értékű index esetén hiba
- Az *x*`[0]` kifejezés értéke üres vektor, amelynek típusa megegyezik *x* típusával
  - Nulláktól eltekintve csak pozitív vagy csak negatív egészeket tartalmazó indexekben a nullák figyelmen kívül hagyása
- Hiba, ha az index vektorban pozitív és negatív értékek is vannak

Vektorok indexelése [ ] operátorral (4)

- Ha az index a vektorral azonos elemszámú logikai vektor, akkor a TRUE értékű elemekkel megegyező indexű elemek kiválasztása
  - Ha az index a vektornál rövidebb logikai vektor, akkor az index vektor elemeinek ismétlése
  - Ha az index a vektornál hosszabb logikai vektor, akkor az indexelt vektor kiegészítése NA értékekkel

Vektorok indexelése [ ] operátorral (5)

- Az index vektorban megjelenhet NA érték
- Elemek értékének kinyerésénél NA értékkel indexelés eredménye megfelelő típusú NA érték (bináris vektorok esetében 0)
- Az *x*[NA] kifejezés értéke egy *x*-szel azonos hosszú, megfelelő típusú NA értékekből álló vektor
- Ha az *x*[*i*] kifejezés értékadásban balértékként jelenik meg, akkor az index vektorban szereplő NA értékek nem választanak ki egyetlen elemet sem helyettesítésre

Vektorok indexelése karakterláncokkal

- Az index mindkét operátor esetében lehet karakter vektor, ekkor a `names` attribútum alapján történik az elemek kiválasztása
- Teljes névegyezés kell egy elem kiválasztáshoz
  - Ha több elem esetén is teljes névegyezés van, a sorrendben első kiválasztása
- Ha elemek kinyerésnél nincs teljes egyezés egyetlen elem nevével sem, akkor `[]` esetén az eredmény NA, `[[]]` esetén hiba
- Elemek kinyerésekor az indexben megjelenő üres karakterlánc (`""`) és a karakter típusú NA érték egyetlen elemet sem választanak ki

Példa vektorok indexelésére

- Az első 10 elem kiválasztása:
  `x[1:10]`
- Minden páros indexű elem kiválasztása:
  `x[1:(length(x)/2)*2]`
- Az összes elem kiválasztása minden 13. elhagyásával:
  `x[-(1:(length(x)/13)*13)]`
- Negatív elemek kiválasztása:
  `x[x<0]`
- Ugyancsak a páros indexű elemeket választja ki:
  `x[c(FALSE,TRUE)]`

Listák (1)

- Olyan vektorok, amelyeknek az elemei tetszőleges objektumok lehetnek
  - Az atomi vektoroktól eltérően az elemek különböző típusúak lehetnek
  - A listák rekurzív jellegű objektumok, az elemeik lehetnek további listák
- Az elemekhez hozzáférés indexeléssel történik, a vektorokhoz hasonlóan
- A `typeof()` függvény lista argumentumokra a `"list"` karakterláncot adja

Listák (2)

- Listákat a `list(...)` függvénnyel lehet létrehozni
  - Tetszőleges számú argumentumot kaphat a függvény, amelyek az elemeket adják meg
  - Az elemeket meg lehet adni *név* = *érték* módon is, ahol *név* azonosító vagy karakterlánc, ezek a nevek alkotják majd a `names` attribútum értékét
    - Ha egy elemhez nem adunk meg nevet, akkor a `names` attribútum értékében a megfelelő elem `""` lesz
    - Ha egyetlen elemhez sem adunk meg nevet, akkor a `names` attribútum értéke `NULL` lesz
  - A függvényt argumentum nélkül meghívva az eredmény üres lista

Listák (3)

- A logikai visszatérési értékű `is.list(`*x*`)` függvénnyel vizsgálható, hogy az argumentum lista-e
- Az `as.list(`*x*`)` függvénnyel lehet objektumokat listává konvertálni
  - Például függvényre alkalmazva egy olyan listát ad, amelynek elemei a formális argumentumok és a függvény törzse

Listák (4)

- Az `unlist(`*x*`)` függvénnyel lehet listákat egyszerűsíteni
  - A függvény a lista elemeit rekurzívan feldolgozva egy atomi vektort próbál konstruálni az alkotó elemekből
  - Ez akkor nem sikerül, ha a lista elemei között nem vektor jellegű elemek (például olyan nyelvi elemek, minta a nevek) szerepelnek
  - Az utóbbi esetben a visszatérési érték egy olyan lista, amelyben végrehajtásra kerültek az elvégezhető egyszerűsítések

Listák (5)

- Listákat a `c(…)` függvénnyel lehet összefűzni
  - Az argumentumok között egyaránt szerepelhetnek atomi vektorok és listák, ezek valamennyi eleme egy-egy komponense lesz az eredmény listának
  - Ha a függvénynek megadjuk a `recursive=TRUE` argumentumot, akkor valamennyi lista argumentum rekurzív feldolgozása, az eredmény az összes elemet tartalmazó atomi vektor lesz

Listák indexelése (1)

- A vektorok indexelésénél elmondottak vonatkoznak a listákra is
- Azonban használható indexelésre a $ operátor is
- Mindhárom operátorral lehet elemeket törölni listákból
  - Ha értékadásban az elemek helyettesítésére használjuk az indexelést, akkor a `NULL` érték megadásával lehet törölni a lista megfelelő elemeit

Listák indexelése (2)

- Listákat indexelni lehet *x*$*index* módon is
  - A kifejezésben *index* azonosító vagy karakterlánc lehet
  - A *names* attribútum alapján történik a megfelelő elem kiválasztása, hasonlóan a `[]` és `[[]]` operátorokhoz
    - Elemek érékének kinyerésénél azonban ezektől eltérően nem szükséges teljes egyezés
      - Ha nincs teljes egyezés egyetlen elem nevével sem, akkor elég prefix egyezés, amennyiben az egyértelmű
      - Teljes egyezés hiányában nem egyértelmű prefix egyezés esetén az eredmény `NULL`
    - Elemek kinyerésekor tehát az indexben az elemek nevét rövidíteni lehet, ha a rövidítés egyértelmű

Listák indexelése (3)

- Az `[ ]` operátor használata esetén az eredmény mindig lista
  - A megfelelő elemekből álló részlista
  - Akkor is, ha az index egyelemű vektor (ilyenkor az eredmény egyelemű lista)
- A lista egyetlen elemét választják ki `[[ ]]` és `$` operátorok
- Az indexben megjelenő NA érték eredménye `NULL`
  - A korábbiak alapján így *x*`[NA]` értéke egy olyan *x*-szel azonos hosszú lista, amelynek minden eleme `NULL`

Példa listák használatára (1)

```
x <- list(longitude=22.38333, latitude=47.95)
x$long
x$lat
y <- list("Makkoshotyka", location=x)
y[[1]] <- "Kocsord"
y[["location"]]$long
y$location$lat
names(y)
names(y)[1] <- "city"
y$city
```

Példa listák használatára (2)

```
z <- list(
   name=list(first.name="John",last.name="Cleese"),
   birth=as.Date("1939-10-27"),
   is.married=TRUE,
   children=2,
   occupation=c("actor","writer","comedian")
)
z$name
z$name$first.name
z$name$last
z[c("name", "birth")]
```

NULL objektum

- Egyetlen NULL objektum létezik, amelyet a `NULL` speciális konstans reprezentál
  - Általa jelezhető egy objektum hiánya (például az, hogy egy kifejezés vagy függvény értéke nem definiált)
- Nincs típusa és nem lehet az attribútumait beállítani
- Az `is.null(`*x*`)` függvény visszaadja, hogy az argumentuma a NULL objektum-e
  - Kizárólag ezt használjuk a NULL objektummal való összehasonlításhoz vagy az `identical(`*x*`, NULL)` kifejezést
- A `typeof(NULL)` kifejezés eredménye `"NULL"`

Kifejezés objektumok (1)

- A kifejezés objektumok elemzett, de nem kiértékelt utasításokat tartalmaznak
  - Maguk az utasítások szintaktikailag helyesek, tokenekből állnak (például literálokból, kulcsszavakból, operátorokból)
- Az `expression(...)` függvénnyel lehet kifejezés objektumokat létrehozni
  - A visszatérési érték egy olyan speciális vektor, amelyre a `typeof()` függvény `"expression"` értéket ad
  - Úgy lehet indexelni őket, mint a listákat

Kifejezés objektumok (2)

- Az `eval(`*x*`)` függvénnyel végezhető el az argumentumként adott kifejezés objektum kiértékelése
  - Meg lehet adni argumentumként azt a környezetet is, amelyben a kiértékelést el kell végezni (ha nem adjuk meg, akkor a hívás helyének környezetét használja a függvény)
  - Ha a kifejezés objektum több kifejezést tartalmaz, valamennyi kiértékelése, és az utolsó értéke a függvény visszatérési értéke

Kifejezés objektumok (3)

- A `D(`*x*, *v*`)` függvény például szimbolikus deriválást végez
  - *x* egy kifejezés objektum, *v* pedig egy karakterlánc (egyelemű karakter vektor), amely megadja, hogy mely változó szerint kell deriválni
- Kifejezés objektumokat lehet használni matematikai formulák megjelenítéséhez ábrákon

Példa kifejezés objektumok használatára

```
e <- expression(1 / (1 + exp(-x)))
D(e, "x")
eval(e, list(x=0))

curve(pnorm(x), -5, 5,
   ylab=expression(P(x)),
   main=expression(
      P(x) == frac(1, sqrt(2 * pi)) * exp(-x^2 / 2)
   )
)
```

Szimbólum objektumok

- A szimbólumok objektumokra hivatkoznak
- Az objektumok neve általában szimbólum
- Az `as.symbol(`*x*`)` és `as.name(`*x*`)` függvényekkel lehet létrehozni szimbólum objektumokat
  - Az argumentum egy karakterlánc lehet
- Az `is.symbol(`*x*`)` és `is.name(`*x*`)` függvények visszaadják, hogy az argumentumuk szimbólum objektum-e
- A `typeof()` függvény `"symbol"` értéket ad, ha az argumentuma szimbólum objektum

Konstansok

- Öt fajta konstans van:
  - logikai
  - numerikus
  - egész
  - komplex
  - karakterlánc
- Speciális konstansok:
  - `NULL`
  - `NA`
  - `Inf`
  - `NaN`

Logikai konstansok

- TRUE és FALSE, amelyek egyelemű logikai vektorok
  - Használható T és F is, amelyek TRUE illetve FALSE értékű globális változók

Numerikus konstansok (1)

- Valós típusúak
- Valamennyi konstans előjel nélküli
  - + és - operátorok!
- Decimális és hexadecimális numerikus konstansokat is használni lehet

Numerikus konstansok (2)

- A decimális konstansok a programozási nyelvekben megszokottak
  - Legalább egy decimális számjegyből és egy opcionális tizedespontból (`'.'`) álló karaktersorozatok, amelyeket `'e'` vagy `'E'` karakter után követhet egy előjeles vagy előjel nélküli, legalább egy hosszúságú decimális számjegyekből álló karaktersorozat
  - Például: `1`, `.5`, `3.141593`, `10e-7`, …

Numerikus konstansok (3)

- Hexadecimális konstansok megadása a `0x` illetve `0X` előtaggal lehetséges
  - Az előtagot a `'0'`, …, `'9'`, `'a'`, …, `'f'` illetve `'A'`, …, `'F'` hexadecimális számjegyek követhetik, amelyekből legalább egy kötelező
    - Például: `0x1FFFF`
  - A hexadecimális számjegyeket `'p'` vagy `'P'` karakter után követheti egy előjeles vagy előjel nélküli, legalább egy hosszúságú decimális számjegyekből álló karaktersorozat (szorzás 2 megfelelő hatványával)
    - Például `0x10p16` = `0x10` × 2^{16}, `0xFp-10` = `0xF` × 2^{-10}

Egész konstansok

- Egész konstans létrehozásához írjuk egy numerikus konstans végére az `L` minősítőt
  - Ügyeljünk arra, hogy kizárólag `'L'` használható, `'l'` nem!
  - Például: `1L`, `1234L`, `1e4L`, `0xffffL`, `0xFp25L`, ...
- Ha a minősítő egy nem egész értékű numerikus konstanst követ, akkor a konstans valós típusú, valamint a rendszer figyelmeztet
- Figyelmeztetést eredményez az is, ha egész értékű minősített konstansban szerepel tizedespont (de az eredmény egész típusú)

Komplex konstansok

- Komplex konstansok megadásához egy numerikus konstans után az `'i'` karaktert kell írni
  - Például: `0i`, `1.5i`, `1e-7i`, `0xFFi`, …
- Nincs valós rész, csak képzetes!
  - Valós és képzetes részből álló komplex számok létrehozása operátorokkal

Karakterlánc konstansok

- Megadásuk `'''` vagy `'"'` karakterek között
  - Például: `"Hello, world!"`
- Nyomtatható karakterekből állhatnak
- Speciális karakterek megadására a programozási nyelvekben megszokott escape szekvenciákat lehet használni, mint például `\n`, `\'`, `\"` vagy `\u`*nnnn*

NULL

- A NULL objektum jelzésére szolgáló speciális konstans

NA (1)

- Hiányzó értéket jelölő speciális konstans („not available”)
- Alapértelmezésben logikai típusú konstans (egyelemű logikai vektor), amelyet azonban tetszőleges típusúvá lehet konvertálni, így tetszőleges típusú vektorban jelölhet hiányzó elemet
  - Kivételt képeznek a bináris vektorok, amelyekhez nincs NA érték
- NA értékeken végzett műveletek eredménye általában NA
  - Kivéve akkor, ha a művelet eredménye az NA értékektől függetlenül meghatározható
  - Lásd például a logikai műveleteket

NA (2)

- Az `is.na(`*x*`)` függvény
  - Visszatérési értéke egy megfelelő elemszámú logikai vektor
  - Az argumentum minden elemének tesztelése egyenként
  - Az eredmény vektor egy eleme TRUE, ha az argumentum megfelelő komponense NA vagy NaN érték
    - Komplex érték NA-ként kezelt, ha a valós vagy a képzetes rész NA vagy NaN
  - Kizárólag ezt használjuk NA érték tesztelésre, mivel például az `NA==NA` kifejezés értéke is NA!
- Az `NA_integer_`, `NA_real_`, `NA_complex_` és `NA_character_` speciális konstansok a megfelelő típusú NA értéket jelölik

Inf (1)

- A végtelent jelölő speciális numerikus konstans
- `Inf`, `-Inf` minden matematikai műveletnél és függvénynél megjelenhet operandusként, paraméterként és eredményként is
- Például `Inf` az értéke az `1/0` kifejezésnek, a `-1/0` kifejezésnek pedig `-Inf`
- Például `as.integer(Inf)` eredménye egész típusú NA érték

Inf (2)

- Az `is.finite(`*x*`)` függvény
  - Visszatérési értéke egy megfelelő elemszámú logikai vektor
  - Az argumentum minden elemének tesztelése egyenként
  - Ha az argumentum nem logikai, valós, egész vagy komplex vektor, akkor az eredmény vektor minden eleme `FALSE`
  - Logikai, valós, egész és komplex vektoroknál az eredményül adott vektorban egy elem `TRUE`, ha az argumentum megfelelő komponense `NA`, `Inf`, `-Inf` és `NaN` értékektől különböző
  - Komplex számok esetében a valós és képzetes részre is teljesülnie kell az előbbi feltételnek

Inf (3)

- Az `is.infinite(`*x*`)` függvény
  - Visszatérési értéke egy megfelelő elemszámú logikai vektor
  - Az argumentum minden elemének tesztelése egyenként
  - Ha az argumentum nem valós vagy komplex vektor, akkor az eredmény vektor minden eleme `FALSE`
  - Valós és komplex vektoroknál az eredményül adott vektorban egy elem `TRUE`, ha az argumentum megfelelő komponense `Inf` vagy `-Inf`
  - Komplex számok esetében elég, ha a valós vagy a képzetes rész közül az egyikre teljesül a feltétel

NaN (1)

- Speciális numerikus konstans („not a number”)
- Azt jelenti, hogy egy művelet nem értelmezett
  - Ez az értéke például a `0/0` vagy az `Inf-Inf` kifejezésnek
- Egész, logikai, komplex típusra konvertáláskor az eredmény megfelelő típusú NA érték
- Minden matematikai operátornál, függvénynél megjelenhet operandusként, paraméterként és eredményként is

NaN (2)

- Az `is.nan(`*x*`)` függvény visszaadja, hogy az argumentuma NaN érték vagy sem
  - Egy megfelelő hosszúságú logikai vektort ad vissza
  - Az argumentum minden elemének tesztelése egyenként
  - Az eredményül adott vektorban egy elem TRUE, ha az argumentum megfelelő komponense NaN érték
    - Komplex érték NaN-ként kezelt, ha a valós vagy képzetes rész NaN
  - Kizárólag ezt használjuk NaN érték tesztelésre vagy az `identical(`*x*`, NaN)` kifejezést, mert a `NaN==NaN` kifejezés értéke NA!

Infix és prefix operátorok (1)

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

Infix és prefix operátorok (2)

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

Infix és prefix operátorok (3)

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

Aritmetikai operátorok (1)

- Kétoperandusú 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, amelyek minden esetben valós eredményt adnak
- Az operandusok lehetnek logikai típusúak, amelyek automatikusan egész típusúvá lesznek konvertálva
  - TRUE érték 1 lesz, FALSE pedig 0

Aritmetikai operátorok (2)

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

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 0)

Aritmetikai operátorok (3)

- 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)

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, a NaN értéket kivéve, amely NA értéket
  - Bináris vektorok esetében a `!`, `&` és `|` operátoroknál nincs konverzió, bitenkénti műveletvégzés történik
  - Karakter vektorok esetén hiba
- Operandusként megjelenhet NA

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

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
  - Például: `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
  - Például: `any(x!=0)`
- Mindkét függvény logikai típusúvá alakítja az argumentumait

Néhány hasznos logikai függvény (2)

- Az `ifelse(`*feltétel*, *igaz*, *hamis*`)` függvény egy *feltétel*-lel megegyező alakú értéket ad, amelynek 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*-évél (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

Néhány hasznos logikai függvény (3)

- Például:

```
ifelse(age >= 18, "adult", "underage")

x <- runif(100, -1, 1)
x <- ifelse(abs(x) > 0.5, x, 0)

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

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

Összehasonlító operátorok (2)

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

Értékadó operátorok (1)

- Az *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
- Az `<-` és `<<-` operátorok jobbról, az `->` és `->>` operátorok pedig balról asszociatívak
- Minden esetben az *érték* kifejezés értéke az értékadó kifejezések értéke
  - Tehát megengedettek például az `z <- y <- x <- 1` és `1 -> x -> y -> z` kifejezések

Értékadó operátorok (2)

- Az `<-` és `->` operátorok esetében az értékadás abban a környezetben történik, amelyben az értékadó kifejezés kiértékelésre kerül
- Az `<<-` és `->>` operátorok az adott nevű szimbólumot keresik abból a környezetből kiindulva, amelyben 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

Értékadó operátorok (3)

- Az *x* `<-` *érték*, *érték* `->` *x*, *x* `<<-` *érték* és *érték* `->>` *x* kifejezésekben *x* lehet egy objektum egy részét megadó kifejezés is
  - Megengedettek például az *x*`[1:10] <- 0` és *x*`[10:19] <- 1:10` értékadások

Az objektumok attribútumai (1)

- A NULL objektum kivételével minden objektumnak lehetnek attribútumai
- Az attribútumok az objektumokhoz tartozó név-érték párok
- Egy objektum attribútumainak halmazként kezelése (egy objektum legfeljebb egy adott nevű attribútummal rendelkezhet)
- Az attribútumok neve tetszőleges karakterlánc, értéke tetszőleges objektum lehet
  - Azonban vannak olyan attribútumok, amelyeknek speciális jelentése van
  - Ezek értékeire előírások vonatkozhatnak

Az objektumok attribútumai (2)

- Az `attributes(`*x*`)` függvénnyel lehet lekérdezni az argumentumként adott objektum attribútumait
  - A visszatérési érték az attribútumok listája
- Az `attributes(`*x*`) <-` *érték* kifejezés beállítja az argumentumként kapott objektum attribútumait
  - *érték* az attribútumokat felsoroló lista, amelynek minden elemét névvel kötelező megadni
  - Az `attributes(`*x*`) <- NULL` az objektum összes attribútumát törli

Az objektumok attribútumai (3)

- Az `attr`(*x*, *név*) függvénnyel lehet lekérdezni egy adott nevű attribútum értékét
  - A második argumentum az attribútum nevét megadó karakterlánc
  - Az attribútum nevével teljes vagy egyértelmű prefix egyezés szükséges (teljes vagy egyértelmű prefix egyezés hiányában `NULL` a visszaadott érték)
- Az `attr`(*x*, *név*) `<-` *érték* kifejezés állítja be az adott nevű attribútum értékét
- Vannak speciális attribútumok értékének lekérdezésére és beállítására szolgáló függvények
  - Speciális attribútumok például: `names`, `dim`, `dimnames`, `class`

A names attribútum (1)

- Lehetővé teszi vektorok, listák elemeinek elnevezését
  - Objektumok kiírásánál fejlécként megjelenik az elemek neve is
  - Indexelésnél lehet a neveket indexként használni (indexelés karakterláncokkal)
- Értéke egy megfelelő elemszámú karakter vektor

A names attribútum (2)

- A `names(`*x*`)` függvény használható a `names` attribútum értékének lekérdezésére
- A `names(`*x*`) <-` *érték* kifejezés beállítja a `names` attribútum értékét
  - *érték* karakter vektorrá konvertálása
  - Ha a karakter vektorrá konvertált *érték* rövidebb *x*-nél, kiegészítése NA értékekkel (hosszabb azonban nem lehet)
  - Üres karakterlánc (`""`) a vektorban azt jelzi, hogy a megfelelő elemnek nincs neve
    - Ezt indexként megadva azonban egyetlen elem sem lesz kiválasztva
  - *érték* lehet speciálisan `NULL`, amely a `names` attribútum törlését jelenti

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 egy kételemű vektor (az első elem a sorok, a második az oszlopok száma)
- Az attribútumot a tömböket és mátrixokat manipuláló függvények, operátorok automatikusan kezelik

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, amelyek szorzata egyenlő *x* hosszával
    - Megadható valós vektor is, amely egésszé 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

A dimnames attribútum (1)

- Tömbök és mátrixok esetén használható a dimenziók elnevezésére
- Arra való, mint a vektorok `names` attribútuma
  - Az attribútum értéke a kiíráskor megjelenő fejléc információt adja meg
  - Valamint lehetővé teszi az indexelést karakterláncokkal

A dimnames attribútum (2)

- Tömböknél és mátrixoknál az attribútum értéke egy a dimenziók számával megegyező elemszámú lista
  - A lista komponenseinek lehetnek nevei (nem kötelező), amelyek a dimenziókat nevezik el
  - A listában valamennyi komponens a megfelelő a dimenzió méretével azonos hosszú karakter vektor lehet vagy pedig `NULL`
  - Ismétlődő értékek megengedettek a lista komponenseiben

A dimnames attribútum (3)

- A `dimnames(`*x*`)` függvény használható az attribútum értékének lekérdezésére
- A `dimnames(`*x*`) <-` *érték* kifejezés beállítja a `dimnames` attribútum értékét
  - *érték* a leírtaknak megfelelő lista lehet vagy `NULL`
    - A listában meg lehet adni tetszőleges típusú vektorokat, valamennyi karakter vektorrá lesz konvertálva
    - Üres lista megadása ekvivalens `NULL` megadásával
    - Ha a listának a dimenziók számánál kevesebb komponense van, akkor kiegészítés `NULL` értékekkel
  - `NULL` megadásával törölni lehet az attribútum értékét

Objektumok hossza

- A `length(`*x*`)` függvény szolgáltatja vektorok, listák, faktorok és egyéb objektumok hosszát
  - Vektorok, listák, faktorok esetében ez az elemek számát jelenti
- A `length(`*x*`) <-` *érték* kifejezés, ahol *érték* egyelemű egész vektor, beállítja az adott vektor vagy objektum hosszát
  - Vektor vagy lista hosszának csökkentése csonkolást eredményez
  - Vektor vagy lista hosszának növelése NA értékekkel kiegészítést eredményez az adott hosszra
- Karakterláncok karaktereinek számának meghatározásához használjuk az `nchar(`*x*`)` függvényt

Mód és tárolási mód (1)

- A `mode(`*x*`)` és `storage.mode(`*x*`)` függvények visszaadják az argumentumként adott objektum módját illetve tárolási módját
  - A két függvény gyakran ugyanazt az értéket adja
  - A visszaadott érték a `typeof(`*x*`)` függvény által szolgáltatott típustól függ
- A `mode(`*x*`) <-` *érték* és `storage.mode(`*x*`) <-` *érték* kifejezések beállítják az adott objektum módját és tárolási módját
  - *érték* a módot vagy tárolási módot megadó karakterlánc

Mód és tárolási mód (2)

- A `mode(`*x*`)` függvény visszatérési értéke néhány kivételtől eltekintve megegyezik a `typeof(`*x*`)` függvény által szolgáltatott típusnevekkel:
  - Az `"integer"` és `"double"` típusok esetében a visszaadott érték a `"numeric"` karakterlánc
  - A `"special"` és `"builtin"` típusok esetében a visszaadott érték a `"function"` karakterlánc
  - A `"symbol"` típus esetében a visszaadott érték a `"name"` karakterlánc
  - A `"language"` típus esetében a visszaadott érték a `"("` vagy `"call"` karakterlánc

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

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 visszakapjuk az alapul szolgáló vektort

Tömbök és mátrixok létrehozása (2)

- Tekintsük például az `x<-1:6` értékadással létrehozott vektort!
  - A `dim(x)<-c(2,3)` értékadás végrehajtása után *x* egy két sorból és három oszlopból álló mátrix:

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

  - A `dim(x)<-c(3,2)` értékadás végrehajtása után *x* egy három sorból és két oszlopból álló mátrix:

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

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)

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 attribútumok értékét szolgáltatják

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

Vektorok és egydimenziós tömbök

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

```
x <- 1:10
y <- x
dim(y) <- c(1,10)
```

  - Az `all(x==y)` kifejezés értéke `TRUE`, mivel *x* és *y* elemei megegyeznek
  - Azonban az `identical(x,y)` kifejezésé `FALSE`, mivel *x* vektor, *y* pedig egy sorból álló mátrix

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`

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 egésszé konvertálása
  - Ha *x* legalább kételemű vektor vagy egydimenziós tömb, akkor egy olyan diagonális mátrix, amelynek főátlójában *x* elemei szerepelnek
  - Ha *x* mátrix, akkor a főátlóban lévő elemekből képzett vektor

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 kezelése)
  - A második „egymás mellé” illeszti az argumentumként adott mátrixokat (vektorok oszlopvektorokként kezelése)
  - Az argumentumként adott mátrixok azonos számú sorból/oszlopból kell hogy álljanak, a vektorok azonban lehetnek rövidebbek (kiegészítés az elemek ismétlésével)

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(`*x*`,` *z*`)` függvény az *x* `%*%` *y* = *z* 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

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)

Tömbök és mátrixok indexelése (2)

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

Tömbök és mátrixok indexelése (3)

- Speciálisan meg lehet adni indexként egy olyan numerikus mátrixot, amelynek a dimenziókkal megegyező számú oszlopa van
  - Nem egész értékek egésszé 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

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

Példa mátrixok indexelésére (1)

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

Példa mátrixok indexelésére (2)

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

Faktorok

- A faktorok olyan speciális vektor jellegű objektumok, amelyek velük azonos hosszú vektorok elemeinek egy osztályozását határozzák meg
  - A fakort alkotó véges számú különböző érték reprezentálja az osztályokat vagy kategóriákat
- A faktor objektumoknak van egy `levels` attribútuma, amelynek értéke egy karakter vektor
  - Ennek elemei adják meg az osztályok vagy kategóriák neveit
  - Ilyen attribútuma tetszőleges objektumnak lehet tetszőleges értékkel
  - Faktorok esetében azonban az attribútum értéke kötelezően egy megfelelő elemszámú karakter vektor kell hogy legyen

Faktorok létrehozása (1)

- Faktorokat a
  `factor(`*x*`, levels=sort(unique.default(`*x*`), na.last=TRUE), labels=levels, exclude=NA, ordered=is.ordered(`*x*`))`
  függvénnyel lehet létrehozni tetszőleges típusú vektorokból
  - Az *x* argumentum egy olyan vektor, amely általában kevés számú különböző értéket tartalmaz (a különböző értékek reprezentálják az egyes kategóriákat)
  - A `levels` argumentum adja meg, hogy az *x* vektorban milyen különböző értékek szerepelhetnek
  - Ha az utóbbit nem adjuk meg, akkor az alapértelmezés egy olyan vektor, amelyben *x* valamennyi különböző eleme növekvő sorrendbe rendezve szerepel (a rendezés módja a környezettől függhet)

Faktorok létrehozása (2)

- A hatékonyabb tárolás érdekében a faktor objektumokat egy egész kódokból álló vektor alkotja a `levels` attribútum értékével együtt
  - A kategóriák neveit a `levels` attribútum adja meg, a kódok ezt a vektort indexelik

Faktorok létrehozása (3)

- A faktor objektum létrehozása az alábbi módon történik:
  - A `levels` argumentumban adott vektor elemei közül törlődnek az `exclude` argumentumban adott vektor elemei
  - Ha *x*[*i*] egyenlő *levels*[*j*]-vel, akkor az eredmény *i*-edik eleme *j* lesz, ha nincs egyezés *levels* egyik elemével sem, akkor pedig NA
  - Ha nem adjuk meg a `labels` argumentumot, akkor a `levels` argumentum értékéből lesz képezve a `levels` attribútum értéke (szükség esetén automatikusan karakter vektorrá konvertálás)
  - Ha megadjuk a `labels` attribútumot, akkor ez egy megfelelő elemszámú karakter vektor lehet, amelynek elemei elnevezik a kategóriákat, ez lesz a `levels` attribútum értéke

Függvények faktorok kezeléséhez (1)

- Az `is.factor(`*x*`)` logikai visszatérési értékű függvénnyel vizsgálható, hogy az argumentumként adott objektum faktor-e
- Az `nlevels(`*x*`)` függvény adja meg az argumentumként adott objektum `levels` attribútumának értékében az elemek számát (azaz az osztályok vagy kategóriák számát)
  - `0`-t ad olyan objektumok esetén, amelyeknek nincs `levels` attribútuma

Függvények faktorok kezeléséhez (2)

- A `levels(`*x*`)` függvény visszaadja `levels` attribútum értékét
- A `levels(`*x*`) <-` *érték* kifejezés a kategóriák átnevezéséhez használható
  - A kifejezésben *érték* egy megfelelő elemszámú, az osztályok vagy kategóriák neveit tartalmazó karakter vektor
  - Az *érték* vektorban megjelenő NA érték a megfelelő kategória törlését jelenti (a kategóriának megfelelő értékek helyettesítése NA értékekkel a faktorban)

Faktorok összehasonlítása

- Faktorok összehasonlításához az `==` és `!=` operátorokat lehet használni
  - Összehasonlítás karakter vektorokkal lehetséges, valamint olyan faktorokkal, amelyeknél a `levels` attribútumban ugyanazok az értékek szerepelnek (nem feltétlenül ugyanabban a sorrendben)

Példa faktorok használatára (1)

```
downloads <- c(6671034, 7326266, 1221041, 14386809,
   427062619,  17959640, 18811435
)
names(downloads) <- c("Firebird", "Celestia",
   "HSQLDB", "JBoss.org", "Azureus",
   "NASA World Wind", "phpBB"
)
lang <- c("C++", "C++", "Java", "Java", "Java",
   "C#", "PHP"
)
langf <- factor(lang)
levels(langf)
unclass(langf)
tapply(downloads, langf, max)
```

Példa faktorok használatára (2)

```
topicf <- factor(c(2, 3, 2, 5, 1, 3, 1), levels=1:5)
levels(topicf) <- c(
   "Communications",
   "Database",
   "Education",
   "Multimedia",
   "Software Development"
)
table(langf, topicf)
```

Példa faktorok használatára (3)

```
uefa <- read.table("uefa.tab", sep="\t")
attach(uefa)
by(uefa, group, summary)
detach(uefa)
```

Adatkeretek

- Olyan speciális listák, amelyeknek a komponensei azonos hosszúak
  - A komponensek vektorok, listák (más adatkeret objektumok is), faktorok és mátrixok lehetnek
  - Mátrixok esetében hossz alatt a sorok számát kell érteni
- Olyan mátrix-szerű struktúrákként tekinthetünk rájuk, amelyekben az oszlopok különböző típusúak lehetnek, és amelyekben az oszlopokat a lista komponensei adják meg
  - A sorok megfigyelések, az oszlopok pedig változók

Sorok és oszlopok elnevezése adatkeretekben

- A `names` attribútum az oszlopok neveit adja meg
  - Ha a létrehozáskor nem adunk egy oszlopnak nevet, akkor az R automatikusan választ megfelelőt
- Minden adatkeret objektumnak van `row.names` attribútuma, amelynek értéke egy karakter vektor
  - Ennek éppen annyi eleme van, amennyi a sorok száma
  - Nem lehet az elemek között ismétlődés és NA értékek sem megengedettek
  - Ha ezt nem adjuk meg, akkor az R automatikusan választ megfelelő értéket az attribútumnak
  - A vektor elemeivel a sorokat lehet indexelni

Adatkeretek létrehozása

- A `data.frame(...)` függvénnyel lehet létrehozni adatkeret objektumokat
- Alkalmas objektumokat (például mátrixokat, listákat) az `as.data.frame(`*`x`*`)` függvénnyel lehet adatkeret objektumokká konvertálni
- Számos olyan függvény van, amelyekkel külső forrásból lehet adatkereteket létrehozni
  - Ilyenek a `read.table()`, a `read.csv()`, `read.csv2()`, `read.delim()`, `read.delim2()` és `read.fwf()` függvények

A `data.frame()` függvény (1)

- A `data.frame(..., row.names=NULL)` függvény adatkeretek létrehozására szolgál
  - Van néhány további hasznos argumentum is, ezeket lásd a dokumentációban
- A függvénynek tetszőleges sok argumentumot meg lehet adni, ezekből a komponensekből képződnek az adatkeret oszlopai
  - Ha egy argumentumot *név* = *érték* módon adunk meg, ahol név azonosító vagy karakterlánc, akkor a rendszer a nevet használja a megfelelő oszlop(ok) elnevezéséhez
  - Egyébként az argumentum alapján határozza meg az oszlopneveket (szükség esetén automatikusan generálja)

A `data.frame()` függvény (2)

- Az oszlopokat szolgáltató argumentumok azonos hosszúak kell hogy legyenek
  - Kivételt képeznek az atomi vektorok, faktorok és az `I(`*x*`)` függvénnyel levédett karakter vektorok (utóbbit lásd a következő oldalon)
  - Szükség esetén ezek azonos hosszra normalizálása az elemek ismétlésével

A `data.frame()` függvény (3)

- A függvénynek átadott karakter vektorok automatikusan faktorokká lesznek konvertálva
  - Ezt úgy lehet megakadályozni, hogy a vektort az `I(`*x*`)` függvénnyel levédve adjuk át a `data.frame(...)` függvénynek
  - Kivételt képez a `row.names` attribútum értékét adó karakter vektor argumentum, amelyből nem lesz oszlop
- Ha mátrixokat, listákat vagy adatkereteket adunk át argumentumokként, akkor ezek oszlopai illetve komponensei egyenként adódnak hozzá a létrehozandó adatkerethez
  - Ezt meg lehet akadályozni úgy, hogy az `I(`*x*`)` függvényt használjuk az argumentum levédésére

A `data.frame()` függvény (4)

- A `row.names` argumentumban egy olyan egész vagy karakter vektort lehet megadni, amelyből a `row.names` attribútum értéke lesz, vagy pedig speciálisan egy egyelemű egész vagy karakter vektort
  - Utóbbi esetben a sorszámával vagy a nevével adott argumentum szolgáltatja a `row.names` attribútum értékét
  - Ez nem is fog megjelenni az adatkeretben oszlopként
  - Kivételt képez az a speciális eset, ha az adatkeret egyetlen sort tartalmaz, ekkor az argumentum a sor nevét adja meg

A `data.frame()` függvény (5)

- A `row.names` argumentum értéke lehet `NULL`, amely az alapértelmezés
  - Ekkor a sorok elnevezése az első olyan komponens alapján történik, amely alkalmas sornevekkel rendelkezik
  - Ha nincs ilyen, akkor a sorok elnevezése egészekkel (számozás egytől)

Példa adatkeret létrehozására

```
countries <- data.frame(
   name=c("Hungary", "Kenya", "Japan"),
   area=c(93030, 580367, 377873),
   oecd.member=c(TRUE, FALSE, TRUE),
   population=c(9905596, 39002772, 127078679),
   capital=I(c("Budapest", "Nairobi", "Tokyo")),
   row.names="name"
)
```

Adatkeretek indexelése (1)

- Az indexelés hasonlóan történik a mátrixok és listák indexeléséhez
  - A `[]`, `[[]]` és `$` operátorokat lehet használni
- Indexelés nem csupán elemek értékének kinyerésére használható, hanem értékadásban is az adatkeret elemeinek helyettesítésére
  - A viselkedés esetenként eltérő lehet elemek kinyerése és értékadás esetében!

Adatkeretek indexelése (2)

- A $ operátorral valamint a `[]` és `[[]]` operátoroknál egyetlen indexet megadva listaként lehet indexelni az adatkereteket
  - Így oszlopokat lehet kiválasztani
- A `[]` és `[[]]` operátoroknál két indexet (sor, oszlop) megadva úgy indexelhetjük az adatkereteket, mint a mátrixokat
  - A `[[]]` operátorral csak egyetlen elemet lehet kiválasztani, a `[]` operátorral egyidejűleg többet is
- Az indexeket kizárólag az `[]` operátornál lehet elhagyni

Adatkeretek indexelése (3)

- Ha egyetlen indexet adunk meg a `[[]]` operátornak, akkor az az adatkeret megfelelő oszlopát kapjuk eredményül
  - Elemek kinyerésénél az eredmény egy vektor vagy `NULL`
  - Az index lehet numerikus típusú (egésszé konvertálás), amely az oszlop számát adja meg
    - Elemek kinyerésénél nem létező oszlop esetén hiba
  - Az index lehet az oszlop nevét megadó karakterlánc
    - Elemek kinyerésénél nem létező oszlop esetén az eredmény `NULL`
  - Például `countries[[1]]`, `countries[["area"]]` is az adatkeret első oszlopát adja, de a `countries$area` kifejezés is

Adatkeretek indexelése (4)

- Ha két indexet adunk meg a `[[]]` operátornak, akkor az az adatkeret megfelelő sorának megfelelő oszlopában lévő értéket jelenti
  - Elemek kinyerésénél az eredmény egy egyelemű vektor vagy `NULL`
  - Az indexek lehetnek numerikus típusúak és karakterláncok, a sor- és oszlopindex lehet különböző típusú
    - Például a `countries[[1,3]]`, `countries[["Hungary", "population"]]` és `countries[["Hungary",3]]` kifejezések mind az első sor harmadik oszlopában lévő értéket adják
  - Elemek kinyerésénél nem létező számú sor vagy oszlop esetén az eredmény hiba, nem létező nevű sor vagy oszlop esetén `NULL`
  - Értékadásban nem létező sor vagy oszlop esetén hiba

Adatkeretek indexelése (5)

- Ha egyetlen indexet adunk meg a `[]` operátornak:
  - Az index numerikus és karakter vektor lehet, az adott oszlopok lesznek kiválasztva, az adott sorrendben
  - Elemek kinyerésénél az eredmény minden esetben egy adatkeret, amely a kiválasztott oszlopokból áll
  - Ilyen módon tetszőleges számú oszlopot ki lehet választani (ugyanaz az oszlop csak kinyerésnél választható ki többször is)
  - Például `countries[c("area","population")]` és `countries[c(1,3)]` egyaránt az első és harmadik oszlopot választja ki

Adatkeretek indexelése (6)

- Ha két indexet adunk meg a `[]` operátornak:
  - Az indexek numerikus és karakter vektorok is lehetnek, a sor- és oszlopindex lehet különböző típusú
  - Az indexek által adott elemek kerülnek kiválasztásra, az adott sorrendben
  - Elemek kinyerésnél ha nem egyetlen elem kerül kiválasztásra, akkor az eredmény a megfelelő elemekből álló adatkeret, egyébként pedig egy egyelemű vektor
    - Például míg a `countries[1,3]` kifejezés értéke egyelemű vektor, addig a `countries[1:2,c(1,3)]` kifejezésé egy adatkeret
  - A két index közül bármelyiket el lehet hagyni, ilyen módon teljes sorokat és oszlopokat lehet kiválasztani
    - Például a `countries[2:3,]` kifejezés a második és harmadik sort választja ki

Adatkeretek indexelése (7)

- Akár új oszlopokat is hozzá lehet adni az adatkeretekhez (új oszlopokkal bővítés a jobb szélen történik)
  - Egy megfelelő értékadásban olyan oszlopot kell megadni, amely még nem szerepel az oszlopok között (ha az oszlopot a számával adjuk meg, akkor közvetlenül az utolsó oszlop után lehet bővíteni)
- Sorokkal is lehet bővíteni az adatkeret objektumot az utolsó sor után
  - Szükség esetén automatikus kitöltés NA értékekkel a közbülső sorokban

Adatkeretek indexelése (8)

- Értékadásban a helyettesítő érték lehet lista, ekkor a lista minden egyes komponense egy-egy oszlop (egy részének) helyettesítésére lesz felhasználva
  - Ha a komponensek a listában rövidebbek a szükségesnél, akkor az elemek ismétlése
- Ha értékadásban csak oszlopokat választunk ki az adatkeret objektumban, akkor megjelenhetnek `NULL` értékek is a listában, amely a megfelelő oszlopok törlését jelentik

Példa adatkeret bővítésére (1)

- Új oszlop hozzáadása az adatkerethez (az alábbiak ekvivalensek):
  - `countries[["gdp.ppp"]] <- c(19800, 1600, 34200)`
  - `countries["gdp.ppp"] <- c(19800, 1600, 34200)`
  - `countries$gdp.ppp <- c(19800, 1600, 34200)`
  - `countries[5] <- list(gdp.ppp=c(19800, 1600, 34200))`

Példa adatkeret bővítésére (2)

- Új sor hozzáadása az adatkerethez (nem ekvivalensek):
  - `countries["Greece",] <- list(131990, TRUE, 10737428,"Athens")`
  - `countries["Greece",1:4] <- list(131990, TRUE, 10737428, "Athens")`
- Különbségek:
  - Az első esetben az új sor valamennyi oszlopába NA-tól különböző érték kerül (szükség esetén a listában adott elemek ismétlésével)
  - A második esetben az új sor első 4 oszlopába kerülnek a listában adott értékek, a többi oszlopba pedig NA

Adatkeretek indexelése (9)

- Adatkeretek vektorokkal történő indexelésnél lehet negatív elemű index vektorokat használni
  - Ez a megfelelő sorok illetve oszlopok elhagyását jelenti az adatkeretből
  - Például a `countries[,c(-1,-3)]` kifejezés az első és harmadik oszlop elhagyását jelenti, `countries[2,-4]` a második sort adja, amelyből elhagyásra került a negyedik oszlop
- Tilos az indexvektorban pozitív és negatív értékeket is használni

Adatkeretek indexelése (10)

- Elemek kinyerésnél adatkereteket vektorokkal indexelve pozitív és negatív indexek között is megjelenhetnek 0 értékek, amelyek figyelmen kívül lesznek hagyva
  - Ha egy index minden eleme 0, akkor az eredmény egy olyan adatkeret, amelynek nincs egyetlen sora vagy oszlopa sem

Adatkeretek létrehozása külső forrásból (1)

```
machine <- read.csv(
   file="http://archive.ics.uci.edu/ml/machine-learning-
databases/cpu-performance/machine.data",
   header=FALSE,
   col.names=c("vendor", "model", "myct", "mmin", "mmax",
      "cach", "chmin", "chmax", "prp", "erp"),
   row.names=2
)
```

Adatkeretek létrehozása külső forrásból (2)

```
bupa <- read.csv(
   file="http://archive.ics.uci.edu/ml/machine-
learning-databases/liver-disorders/bupa.data",
   header=FALSE,
   col.names=c("mcv", "alkphos", "sgpt", "sgot",
      "gammagt", "drinks", "selector"),
   colClasses=list(selector="factor")
)
```

Adatkeretek létrehozása külső forrásból (3)

- Az első argumentumként adott adatkeret oszlopait a `transform(`*x*`, ...)` függvénnyel transzformálhatjuk
  - A további argumentumok *név=érték* alakú kifejezések
  - Az adatkeret adott nevű oszlopai lesznek helyettesítve, ha nincs egy adott nevű oszlop, akkor pedig az adatkeret bővítése
- Például:

```
Orange <- transform(Orange, age = age + 1)
mtcars <- transform(mtcars, am=factor(am))
transform(mtcars, category=factor(
  ifelse(mpg > 20 & wt > 2.5, "A", "B")
))
```

Adatkeretek és listák csatolása a keresési útvonalhoz (1)

- Az `attach(`*x*`, pos=2)` függvénnyel lehet adatkereteket és listákat a keresési útvonalhoz csatolni
  - Az *x* argumentum adatkeret, lista és környezet is lehet
  - Ezután az *x* objektum komponenseit ideiglenesen a nevükkel egyező nevű változókon keresztül lehet elérni
  - Van néhány további argumentum is, ezeket lásd a dokumentációban

Adatkeretek és listák csatolása a keresési útvonalhoz (2)

- Az `attach()` függvény alapértelmezésben a keresési útvonal második elemeként szúrja be az argumentumként adott objektumot
  - Azaz a globális környezet után, valamint az összes előzőleg betöltött csomag és a keresési útvonalhoz csatolt objektum elé
  - Ezt lehet szabályozni a `pos` argumentum megadásával (azonban `pos` értéke nem lehet 1)

Adatkeretek és listák csatolása a keresési útvonalhoz (3)

- Az `attach()` függvény egy olyan új környezet hoz létre, amelybe lemásolódnak az argumentumként adott lista vagy adatkeret komponensei, és ez a környezet lesz hozzáadva a keresési útvonalhoz!
- A továbbiakban a felhasználó a másolatokkal dolgozik, tehát a komponensekkel egyező nevű változókon keresztül nem tudja megváltoztatni az eredeti objektumot!
  - Értékadásban az `<<-` vagy `->>` operátort használva a másolatokat lehet megváltoztatni
  - Az `<-` vagy `->` operátorral történő értékadás pedig a globális környezetben hoz létre egy új változót

Adatkeretek és listák csatolása a keresési útvonalhoz (4)

- Egy keresési útvonalhoz csatolt objektumot a `detach(`*x*`, pos=2)` függvénnyel lehet a keresési útvonalról eltávolítani
  - Meg lehet neki adni argumentumként az útvonalról eltávolítandó objektum nevét mint azonosítót, és mint karakterláncot is
  - Az eltávolítandó objektum megadható a sorszámával is a keresi útvonalban
  - Argumentum nélkül meghívva a keresési útvonal második elemét távolítja el

Példa adatkeretek és listák keresési útvonalhoz csatolására

```
countries$population
attach(countries)
population
ls()
population <- population + 1
ls()
rm(population)
population <<- population + 1
countries$population <- population + 1
detach()
countries$population
```

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, amely 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

Környezetek (2)

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

Globális környezet

- 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

Példa környezetek használatára

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

Keresési útvonal

- Környezeteket tartalmaz, amelyekben 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 visszatérési értéke egy a keresési útvonalat megadó karakter vektor
- Az `attach()`, `detach()` és `library()` függvényekkel lehet manipulálni a keresési útvonalat

Függvények

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

Függvények definiálása

- Egy névtelen függvényt definiál az alábbi kifejezés:

  `function(`*arglista*`)` *törzs*

Formális argumentumlista

- Vessző karakterekkel elválasztott argumentumok alkotják, amelyek 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

Függvények törzse (1)

- 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

Függvények törzse (2)

- Kizárólag függvény törzsében használható a logikai visszatérési értékű `missing(`*x*`)` függvény
  - Argumentumként a tartalmazó függvény egy formális argumentumának nevét lehet megadni
  - Azt szolgáltatja, hogy az adott formális argumentumhoz meg lett-e adva aktuális argumentum a hívás során
    - TRUE értéket ad akkor, ha a paraméterkiértékelés során a formális argumentumnak nem felelt meg egyetlen aktuális argumentum sem és a függvényhívást megelőzően a törzsben nem lett módosítva a formális paraméter értéke

Példa függvények definiálására (1)

```
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)
}
```

Példa függvények definiálására (2)

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

y <- function(w, x, f) {
  if (missing(f))
    f <- function(z) 1 / (1 + exp(-z))
  else
    if (! is.function(f))
      stop("invalid f argument")
  f(sum(w * x))
}
```

Függvényhívás (1)

- Az alábbi módon 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, ha a hívás függvény törzsében történik

Függvényhívás (2)

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

Példa függvényhívásokra

```
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)
```

Rekurzió (1)

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

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 átnevezése
rm(h)
hanoi(3)   # Hiba, mivel nincs h() függvény!
```

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)
hanoi(3)
```

Rekurzió (4)

- Rekurzió helyes megvalósítása:

```
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")
}
```

Függvények és környezetek (1)

- Egy függvény környezete az a környezet, amely a függvény létrehozásakor aktív volt
  - Ennek a környezetnek valamennyi szimbólumát látja a függvény
- A parancssorban létrehozott függvények környezete ilyen módon a globális környezet
- Az `environment(`*`f`*`)` függvény az argumentumként adott függvény környezetét szolgáltatja

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

Függvények és környezetek (3)

- A függvények környezetét szemléltető példa:

```
rm(x)
f <- function() {
  x <- 1
  g <- function(y) x + y
  return(g)
}
h <- f()
environment(f)
environment(h)
ls(envir=environment(h))
get("x", envir=environment(h))
```

Paraméterkiértékelés (1)

- Az aktuális paraméterek formális paraméterekhez hozzárendelése név szerinti illetve 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 argumentumszám

Paraméterkiértékelés (2)

- Az aktuális és formális argumentumok megfeleltetése az alábbiak szerint három lépésben történik:
  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, amelynek nincs alapértelmezett értéke

Paraméterkiértékelés (3)

1. 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

Paraméterkiértékelés (4)

2. Részleges névegyezés alapján:
   - Az előző lépésben formális argumentumoknak nem megfeleltetett névvel megadott aktuális argumentumok mindegyikéhez olyan formális argumentumot keresni a megmaradtak közül, amelynek 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

Paraméterkiértékelés (5)

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

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
  - Ne adjunk aktuális paraméterként olyan mellékhatásos kifejezéseket, mint például az értékadás

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

Függvények vektorizálása

- 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, amely azonban vektorokra is alkalmazható
  - Vektorokra alkalmazás *f* elemenkénti alkalmazását jelenti

Példa függvények vektorizálására

- A korábban definiált függvényeket felhasználó példák:

```
Vectorize(hanoi)(1:30)
Vectorize(ackermann)(0:3, 0)
outer(0:3, 0:3, Vectorize(ackermann))
outer(1:10, 1:10, Vectorize(gcd))
```

Összetett utasítás

- Blokknak vagy összetett utasításnak nevezik az alábbit:

  { *kifejezés*$_1$; *kifejezés*$_2$; … ; *kifejezés*$_n$ }

  - Az összetett utasításban tetszőleges kifejezéseket meg lehet adni, amelyeket el lehet választani újsor 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 újsor karakter

Vezérlési szerkezetek

- Elágaztató utasítások:
  - `if` (*kifejezés*$_1$) *kifejezés*$_2$ `else` *kifejezés*$_3$
  - `if` (*kifejezés*$_1$) *kifejezés*$_2$
- Ciklusszervező utasítások:
  - `repeat` *kifejezés*
  - `while` (*kifejezés*$_1$) *kifejezés*$_2$
  - `for` (*név* `in` *kifejezés*$_1$) *kifejezés*$_2$
  - `break`
  - `next`

Az if utasítás (1)

- Használat:

  `if (` *kifejezés*$_1$ `)` *kifejezés*$_2$ `else` *kifejezés*$_3$

  `if (` *kifejezés*$_1$ `)` *kifejezés*$_2$

  - *kifejezés*$_1$ értéke logikai vagy azzá konvertálható vektor lehet, amelynek első eleme NA értéktől különböző kell hogy legyen
  - *kifejezés*$_2$ és *kifejezés*$_3$ tetszőleges kifejezések
  - Figyelmeztetést eredményez, ha *kifejezés*$_1$ értéke egynél hosszabb vektor (csak az első elem lesz felhasználva)

Az if utasítás (2)

- Végrehajtáskor először kiértékelődik az első kifejezés
  - Ha *kifejezés*$_1$ értéke olyan logikai vektor, amelynek első eleme `TRUE`, akkor kiértékelődik *kifejezés*$_2$ és annak értéke az utasítás visszatérési értéke
  - Egyébként
    - Ha van `else` ág, akkor kiértékelődik *kifejezés*$_3$ és annak értéke az utasítás visszatérési értéke
    - Ha nincs `else` ág, akkor `NULL` az utasítás visszatérési értéke

Az if utasítás (3)

- Ha az `if` utasítás nem összetett utasításban szerepel, akkor az `else` egy sorban kell hogy szerepeljen *kifejezés*$_2$-vel
  - Speciálisan ha *kifejezés*$_2$ összetett utasítás, akkor a végét jelző `'}'` és az `else` között nem megengedett újsor karakter
- Az `if` utasításokat tetszőleges mélységig egymásba lehet ágyazni

Példa az if utasítás használatára

```
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
}
```

A repeat utasítás

- Használat:

  `repeat` *kifejezés*

  - Az utasításban szereplő kifejezés tetszőleges lehet
  - Végtelen ciklus megvalósítására szolgál
  - A kifejezés kiértékelése újra és újra, amelyet 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)
  - Az utasítás visszatérési értéke `NULL`

Példa a repeat utasítás használatára

- Faktorizáció (Pollard ρ-heurisztikája):

```
pollard <- function(n) {
   i <- 1
   x <- sample(0:(n-1), size=1)
   y <- x
   k <- 2
   repeat {
      i <- i + 1
      x <- (x^2 - 1) %% n
      d <- gcd(y - x, n)
      if (d != 1 && d != n)
         print(d)
      if (i == k) { y <- x; k <- 2 * k }
   }
}
```

A while utasítás (1)

- Használata:

  `while` (*kifejezés*$_1$) *kifejezés*$_2$

  - Az utasításban szereplő *kifejezés*$_1$ kifejezésre pontosan ugyanazok a feltételek kell hogy teljesüljenek, mint az `if` utasítás esetében
  - *kifejezés*$_2$ hasonlóan tetszőleges kifejezés lehet
  - A programozási nyelvekben megszokott kezdőfeltételes ciklus

A while utasítás (2)

- Végrehajtáskor először kiértékelődik az első kifejezés
  - Ha *kifejezés*$_1$ értéke olyan logikai vektor, amelynek első eleme TRUE, akkor kiértékelődik *kifejezés*$_2$, majd újból *kifejezés*$_1$, …
  - Az utasítás végrehajtása befejeződik akkor, ha *kifejezés*$_1$ kiértékelése FALSE értéket eredményez
- Az utasítás visszatérési értéke NULL

Példa a while utasítás használatára (1)

- Naiv prímteszt:

```
is.prime <- function(n) {
   if (length(n) != 1) stop("invalid argument")
   n <- as.integer(n)
   if (n == 2) return(TRUE)
   if (n == 1 || n %% 2 == 0) return(FALSE)
   m <- 3
   while (m * m <= n) {
      if (n %% m == 0) return(FALSE)
      m <- m + 2
   }
   TRUE
}
```

Példa a while utasítás használatára (2)

- Legnagyobb közös osztó meghatározása (euklideszi algoritmus):

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

Példa a while utasítás használatára (3)

- Pozitív egész kettes számrendszerbeli alakjának meghatározása:

```
as.binary <- function(n) {
  ret <- integer()
  while (n > 0) {
    m <- n %/% 2
    d <- as.integer(n - 2 * m)
    ret <- append(d, ret)
    n <- m
  }
  ret
}
```

A for utasítás (1)

- Használata:

  `for (` *név* `in` *kifejezés*$_1$ `)` *kifejezés*$_2$

  - Az utasításban szereplő *kifejezés*$_1$ olyan kifejezés lehet, amelynek értéke atomi vektor, lista, kifejezés objektum vagy `NULL`
  - *kifejezés*$_2$ tetszőleges kifejezés lehet

A for utasítás (2)

- Végrehajtáskor az adott nevű változó sorban felveszi értékül *kifejezés*$_1$ értékének elemeit és minden elemre kiértékelődik *kifejezés*$_2$
  - Csak egyszer kerül kiértékelésre *kifejezés*$_1$
  - Ha *kifejezés*$_1$ értéke `NULL`, akkor egyszer sem értékelődik ki *kifejezés*$_2$
  - *kifejezés*$_2$-ben meg lehet változtatni a változó értékét, de az ettől függetlenül felveszi *kifejezés*$_1$ értékének minden elemét
- Az utasítás visszatérési értéke `NULL`

A for utasítás (3)

- Az utasítás végrehajtása után mellékhatásként a változó értéke az lesz, amelyet a kiértékelés során utoljára felvett
  - Speciálisan a változó értéke `NULL` lesz akkor, ha *kifejezés*$_1$ értéke `NULL`

Példa a for utasítás használatára (1)

- Moduláris hatványozás:

```
ModularPower <- function(a, b, n) {
   c <- 0
   d <- 1
   for (digit in as.binary(b)) {
      c <- 2 * c
      d <- (d * d) %% n
      if (digit == 1) {
         c <- c + 1
         d <- (d * a) %% n
      }
   }
   d
}
```

Példa a for utasítás használatára (2)

```
findRoot <- function(f, a, b, eps=1e-4, maxit=40) {
   fa <- f(a)
   fb <- f(b)
   if (fa * fb >= 0) stop("f(a) * f(b) must be negative")
   root = if (fa < 0) { dx = b - a; a }
      else { dx = a - b; b }
   for (j in 1:maxit) {
      fmid <- f(xmid <- root + (dx <- dx / 2))
      if (fmid <= 0) root <- xmid
      if (abs(dx) < eps || fmid == 0) return(root)
   }
}
```

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

Példa a break és next utasítások használatára

```
env <- new.env()
repeat {
   line <- readLines(n=1)
   if (length(line) == 0) break
   line <- gsub("(^[[:space:]]+)|([[:space:]]+$)", "", line)
   if (line == "") next
   if (length(grep("^QUIT$", line, ignore.case=TRUE)) == 1)
      break
   try(
      print(
         eval(parse(text=line), envir=env)
      )
   )
}
```

A switch függvény (1)

- Használata:

  `switch(`*kifejezés*`, ...)`

  - Az első argumentumként megjelenő *kifejezés* értéke egyelemű numerikus vagy karakter vektor lehet
  - Ezt tetszőleges számú további aktuális argumentum követheti, amelyek értéke tetszőleges lehet
  - A végrehajtás során először kiértékelődik az első argumentumként adott *kifejezés*, majd a következő módon folytatódik a végrehajtás

A switch függvény (2)

- Ha *kifejezés* értéke egyelemű numerikus vektor:
  - Jelölje *k* ennek egésszé konvertált értékét, *n* pedig a függvény aktuális argumentumainak számát!
  - Ha $1 \leq k \leq (n - 1)$ teljesül, akkor kiértékelődik a $(k + 1)$. aktuális argumentum, amelynek értéke a függvény visszatérési értéke
  - Egyébként `NULL` a függvény visszatérési értéke

A switch függvény (3)

- Ha kifejezés értéke karakterlánc:
  - Ha van olyan argumentum, amelynek a címkéje megegyezik a karakterlánccal, akkor az első ilyen argumentum értéke a visszatérési érték
  - Ha nincs ilyen argumentum, akkor a visszatérési érték az első címke nélküli argumentum értéke, ilyen hiányában pedig `NULL`

Példa a switch függvény használatára (1)

```
NumberOfDays <- function(year, month) {
   switch(month,
      31,
      if (((year %% 4 == 0) && !(year %% 100 == 0))
         || (year %% 400 == 0)) 29 else 28,
      31, 30, 31, 30, 31, 31, 30, 31, 30, 31
      )
}
```

Példa a switch függvény használatára (2)

```
is.prime <- function(n, algorithm="AKS") {
   algorithm <- match.arg(algorithm, c("naive",
      "Fermat", "MillerRabin", "AKS"))
   switch(algorithm,
      naive=NaivePrimalityTest(n),
      Fermat=FermatPseudoprimeTest(n),
      MillerRabin=MillerRabinPseudoprimeTest(n),
      AKS=AKSPrimalityTest(n)
   )
}
```

Adatkészletek (1)

- Az adatkészletek objektumokat szolgáltatnak, amelyek reprezentációit állományok tartalmazzák
  - Az adatkészlet betöltése után állnak rendelkezésre az objektumok
- Általában csomagok részét képezik
- Az R csomagok számos beépített adatkészletet tartalmaznak
  - Adatokat szolgáltatnak, amelyeket fel lehet használni például tesztelésre

Adatkészletek (2)

- A `data(..., list=character(0), package=NULL)` függvénnyel lehet adatkészleteket betölteni és listázni a rendelkezésre álló adatkészleteket
  - Van néhány további argumentum is, ezeket lásd a dokumentációban
  - Ha nem adunk meg a függvénynek betöltendő adatkészletet, akkor a betöltött csomagokból valamint az állományrendszerben az aktuális könyvtárból elérhető adatkészleteket listázza
  - Adatkészletek betöltéséhez a függvénynek meg kell adni a betöltendő adatkészletek nevét mint azonosítót vagy karakterláncot (tetszőleges számú argumentum megadható), és/vagy a `list` argumentum értékéül az adatkészletek neveit tartalmazó karakter vektort

Adatkészletek (3)

- A `help(`*x*`)` függvénnyel lehet hozzáférni egy adatkészletek dokumentációjához, argumentumként annak nevét mint azonosítót vagy karakterláncot megadva

Adatkészletek (4)

- Az adatkészleteket olyan állományok tárolhatják, amelyeknek a neve az alábbi módon végződik:
  1. `'.R'` vagy `'.r'`
  2. `'.Rdata'` vagy `'.rda'`
  3. `'.tab'`, `'.txt'` vagy `'.TXT'`
  4. `'.csv'` vagy `'.CSV'`
- Azaz jelenleg 4 különböző állományformátum támogatott
- A `'.csv'`, `'.txt'` és `'.tab'` állományokat tömörítve is tárolni lehet (gzip, bzip2, xz)

Adatkészletek (5)

- Az 1. fajta állományok a `source()` függvénnyel lesznek beolvasva
  - Ezek tehát R kódot tartalmaznak
  - Beolvasás közben ideiglenesen az állományt tartalmazó könyvtár lesz az aktuális munkakönyvtár
- A 2. fajta állományok a `load()` függvénnyel lesznek betöltve
  - Ezek a `save()` függvénnyel elmentett objektumokat tartalmaznak
- A 3. fajta állományok egy `read.table(..., header=TRUE)` függvényhívással lesznek beolvasva
  - Ezek táblázatos formában tartalmaznak adatokat
- A 4. fajta állományok egy `read.table(..., header=TRUE, sep=";")` függvényhívással lesznek beolvasva
  - Ezek szintén táblázatos formában tartalmaznak adatokat

Adatkészletek (6)

- A betöltés során az R olyan állományokat keres, amelyek neve az adatkészlet nevével kezdődik és ezt a felsorolt végződések valamelyike követi
  - Ha egy adatkészlethez több állományt is talál az R, akkor a sorrend szerinti elsőt választja
    - Ha van olyan állomány, amelynek a neve `'.R'`-re vagy `'.r'`-re végződik, akkor azt, …

Adatkészletek (7)

- A keresés a betöltött, azaz a keresési útvonalhoz adott csomagokban történik és az aktuális munkakönyvtár `data` alkönyvtárában
  - A `data` könyvtárban elhelyezett állományokat össze lehet csomagolni egy `Rdata.zip` nevű állományba, így helyet lehet megtakarítani (a ZIP állomány maradjon a `data` könyvtárban)
  - A `data` könyvtárban a ZIP állomány mellé el kell helyezni egy `filelist` nevű szöveges állományt, amely a benne összecsomagolt állományok nevét sorolja fel (soronként egy állománynév)

Adatkészletek (8)

- Az első két fajta adatkészlet betöltése több változót is létrehozhat a betöltést végző környezetben, amelyek neve tetszőleges lehet
- A másik két fajta adatkészlet betöltése egyetlen változót hoz létre a betöltést végző környezetben, amelynek a neve megegyezik az adatkészlet nevével

Példa adatkészletek használatára

```
data(package="MASS")
help(quine, package="MASS")
data(quine, package="MASS")
data(list=c("galaxies", "shuttle"), package="MASS")
shuttle
summary(shuttle)
rm(list=ls())
library(MASS)
cats
ls()
data(cats)
ls()
summary(cats)
attach(cats)
```