Control flow

Tamas Kadlecsik

February 25, 2016

Vezérlés

A scriptjeinket elképzelhetjük úgy, mint feldolgozóüzemeket. Nagy mennyiségben beérkezik a feldolgozandó anyag, például hús, érc, stb. A lényeg, hogy több különböző komponensre akarjuk szétszedni, majd értékes árut szeretnénk létrehozni. Esetünkben az alapanyag valamilyen nyers adat. A feldolgozást végző különböző (forgó)berendezések, például malmok, reaktorok, darálók, őrlők az úgynevezett **ciklusok** [for]. A betáplált adatok azonos minőségű részhalmazain újra, és újra elvégzik ugyanazt a műveletet. Nekünk csupán arról kell gondoskodnunk, hogy a késztermék helyére minden ciklus után friss alapanyag kerüljön. Semmilyen eljáráshoz nem elegendőek azonban maguk a gépek. Azt, hogy egy adott dolog a képbe kerülhet-e, valamilyen mechanizmusnak ellenőriznie kell. Ezek lesznek a **feltételes vezérlő utasítások** [if].

Hogy egy kicsit mindennapibb példával éljünk:

Amikor tésztát gyúrunk, szükségünk van tojásra, lisztre, vízre. Azonban mielőtt összekevernénk a szükséges mennyiséget, leellenőrizzük, hogy a tojásaink nem zápultak-e meg, illetve, hogy nem zsizsikes-e a lisztünk.

Ezek után addig gyúrjuk, amíg megfelelő állagú tésztát nem kapunk. Ha esetleg túl híg, újabb adag lisztet adunk hozzá, ha túl sűrű, kicsit felvizezzük.

Ezt egy gépnek a következő R pszeudokóddal tudnánk elmagyarázni:

```
flour <- read.table("a bag a flour")
water <- read.table("0.5 l of water")
eggs <- read.table("2 eggs")
if("maggots" %in% flour) flour <- read.table("new bag of flour")
if(is.spolied(eggs)) eggs <- read.table("2 other eggs")
```

Ezzel elő is készültünk a főzéshez. Persze a kód nem teljes, hiszen ezt addig kéne ismételgetnünk, amíg nem találunk használható tojásokat ÉS használható lisztet.

Ezek után addig kell gyúrnunk, amíg megfelelő állagú nem lesz:

```
ready <- F
while(!ready){
    dough <- knead(flour, water, eggs)
    if (dough["consistency"] -- "ready") ready <- T
}
write.table(dough, "dough.csv")
```

Természetesen ezt is pontosíthatnánk még, de nem célunk pontos tésztagyúróprogram írása. A következőkben szépen végignézzük a fenti kódot, illetve írunk használhatót is.

Előre szeretném leszögezni, hogy noha zárójelet látunk utánuk, sem az if, sem a while, sem a többi ciklus nem függvény! Függvényhívás után ugyanis nem állhat kód blokk! Vagyis függvényhívás után sosem látunk "{" karaktert!

Feltétes utasítások

Logikai operátorok

Kódunk legelején a már ismert módon beolvastuk a nyers adatot, vagyis a tésztánk alapanyagait.

```
flour <- read.table("a bag a flour")
water <- read.table("0.5 l of water")
eggs <- read.table("2 eggs")
if("maggots" %in% flour) flour <- read.table("new bag of flour")
if(is.spolied(eggs)) eggs <- read.table("2 other eggs")
```

Az **if kitétel** utáni zárójelben található `["maggots" %in% flour]` logikai kifejezésben az %in% a logikai operátor. Logikai operátorból igen sokféle van, azonban a leggyakoribbak:

Név	Operátor
Vektoriális **AND**	`&`
Logikai **AND**	`&&`
Vektoriális **OR**	`|`
Logikai **OR**	`||`
NOT	`!`
EQUALS TO	`==`
NOT EQUALS TO	`!=`
LESS / GREATER THEN	`<` / `>`
LESS / GREATER THEn OR EQUALS TO	`<=` / `>=`
INCLUDES	`%in%`

Az **AND**, **OR** és **NOT** operátoraink közvetlenül a Bool algebrából lettek kölcsönvéve, így könnyedén írhatunk rájuk igazság táblát:

AND:

A	B	A & B
T	T	T
T	F	F
F	T	F
F	F	F

OR:

A	B	A | B
T	T	T
T	F	T
F	T	T
F	F	F

NOT:

A	!A
F	T
T	F

Magyarázatra szorul még a vektoriális, és logikai operátor. A vektoriális operátorok két egyenlő hosszúságú logikai vektoron végeznek ELEMENKÉNT logikai műveletet.

```
l1 <- c(TRUE,TRUE,FALSE,TRUE,FALSE)
l2 <- c(FALSE,TRUE,FALSE,TRUE,TRUE)
l1 & l2
```

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

```
l1 | l2
```

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

Ezzel szemben a logikai operátor minden esetben biztosítja, hogy csakis egy logikai eredménye legyen a vizsgálatnak. Esetünkben a két vektornak csupán az első elemeit fogja összehasonlítani.

```
l1 && l2
```

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

```
l1 || l2
```

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

A tagadás (**NOT**) operátor logikai vektorokon van értelmezve. Mivel ha csupán egyetlen értéket rendelünk egy változóhoz akkor is vektort kapunk, csupán egyeleműt, így nincs értelme külön vektoriális és logikai tagadásról beszélni. A kettő egy és ugyanaz.

```
!l1
```

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

```
l1 <- TRUE
!l1
```

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

Noha az esetek döntő többségében if kitételen belül is működnek a vektoriális operátorok, ezek használata azonban nem szerencsés, így ha lehet, szokjunk rá, hogy megfelelő helyen a megfelelő operátort használjuk!

A matematikai relációjelekkel a szokásos módon tudjuk megvizsgálni két változó, vagy egy változó és egy konstans viszonyát.

```
x <- 4
x > 3
```

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

```
x < 3
```

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

```
x <= 3
```

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

```
x >= 4
```

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

```
x == 4
```

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

```
x != 4
```

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

```
x == 15
```

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

```
x != 15
```

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

A tartalmazás operátorral egyszerűen megvizsgálhatjuk, hogy valamely vektor, mátrix, vagy bármely összetett típusú objektum tartalmazza-e a keresett elemet vagy elemeket.

```
'apple' %in% c('apple', 'banana')
```

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

```
10 %in% c(1,5,25)
```

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

```
c(1,5) %in% c(1,5,25)
```

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

```
c(1,2) %in% c(1,5,25)
```

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

Figyeljünk, hogy az `%in%` operátort közvetlenül tagadni nem tudjuk! Azonban a matematikai logika (a.k.a. Boole algebra) szabályai itt is alkalmazhatóak, a műveleti sorrend, és a zárójelek használata megegyezik a gimnáziumban tanulttal. Amennyiben már nincs meg a gimis matekkönyvünk, úgy a wikipédia Boole algebrára vonatkozó szócikke segíthet eligazodni. Az `%in%` operátor eredményének tagadásának egyik módja a következő:

```
!(10 %in% c(1,5,25))
```

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

Logikai függvények

```
if(is.spolied(eggs)) eggs <- read.table("2 other eggs")
```

Fenti sorunk, mellyel megvizsgáltuk, hogy a tojásaink romlottak-e, egy logikai függvény. Vagyis egy olyan függvény, melynek visszatérési értéke egy logikai érték. Igen sok ilyen van, a konkrét típus vizsgálatok mindegyikére találunk függvényt:

```
is.vector()
is.numeric()
is.integer()
is.logical()
is.complex()
is.data.frame()
is.list()
```

A két legfontosabb azonban az:

```
is.null()
is.na()
```

A null egy külön típus. Jelentése: semmi. Ahogy a matematikában a 0 a semmi SZÁMSZERŰ reprezentációja, úgy a `NULL` az informatikában a semmi informatikai reprezentációja. Esetünkben a 0 nem semmi, hiszen hordoz információt. A `NULL` azt jelképezi, hogy valamiről nem áll rendelkezésre információ. Amolyan üres memóriaterület. Ha egy változónak nincs értéke, akkor a típusa NULL.

Ezzel szemben az `NA` (Not Available) a "nincs adat" reprezentációja. Ez a statisztikai semmi. Ha nincs mért adatunk, statisztikailag nincs semmink, azonban fontos ezen adatok nyomon követése is. Az, hogy nem mértünk semmit (mert tök részegek voltunk, és nem bírtuk megfogni még a pipettát sem), nem egyenlő azzal, hogy 0 értéket mértünk (mert kimutatási határ alatt volt a mérendőnk). Információ értéke azonban ennek is van!

Az `NA` nem összekeverendő a `NaN értékkel`, ami a "Not a Number" kifejezés rövidítése. Bővebb információért konzol: `?NaN`.

Logikai indexelés

Adott data.frame celláit a következő módon tudjuk elérni: `data.frame[sor, oszlop]` Például a házi feladat sorai, és oszlopai:

```
experiment[2,]
```

visszaadja az experiment második sorát

```
experiment[,3]
```

visszaadja az experiment második sorát

```
experiment$died
```

és

```
experiment["died"]
```

hívások.

Amennyiben adott sorokat szeretnénk vektorokat is megadhatunk indexként. Emlékeztetőül, a vektorok létrehozására:

```
1:5 == c(1,2,3,4,5)
```

Vagyis, ha az első 5 sort szeretnénk:

```
experiment[1:5,]
```

első 3 oszlop:

```
experiment[,1:5]
```

első 5 oszlop, első 5 sora

```
experiment[1:5,1:5]
```

de ha csak az 1., 3., és 5. sorok kellenek:

```
experiment[c(1,3,5),1:5]
```

és csak az első és 5. oszlop:

```
experiment[c(1,3,5),c(1,5)]
```

Miért beszélünk erről a logikai operátoroknál?

Nézzük meg mi történik, ha

```
experiment[,c(F,T)]
```

Vagyis kiegészítette a mintázatot 5-ig, tehát ez a hívás megegyezik a következővel:

```
experiment[,c(F,T,F,T,F)]
```

Tehát

```
experiment[T,T,F]
```

Ugyanaz, mint:

```
experiment[T,T,F,T,T]
```

Az utolsó F már nem fér ki, mivel csak 5 oszlopunk van. Ugyanez sorokkal is működik:

```
experiment[c(F,T), c(F,T)]
```

Megkaptuk az experiment minden második sorát, és oszlopát.

Most nézzük meg mi történik, ha a data.frame-et logikai vizsgálatnak vetjük alá:

```
experiment$treatment == "control"
```

Kapunk egy vektort amely megadja, hogy az adott sorban a logikai vizsgálat eredménye TRUE, vagy FALSE. Ezt felhasználva tudunk szűrni a a táblázatunk sorai között:

```
filter.vector <- experiment$treatment == "control"
experiment[filter.vector,]
```

Ezt egybeírva:

```
experiment[experiment$treatment == "control",]
```

Természetesen több oszlopot is vizsgálhatunk egyszerre:

```
filter.vector <- experiment$treatment == "control" & experiment$diet == "control"
experiment[filter.vector,]
experiment[experiment$treatment == "control" & experiment$diet == "control",]
```

Vezérlési utasítások

Amennyiben egy adott kód lefutására csak bizonyos feltételek teljesülése esetén szükséges, akkor logikai vezérlési utasítást, más néven **if kikötést** alkalmazunk.

```
if("maggots" %in% flour) flour <- read.table("new bag of flour")
if(is.spolied(eggs)) eggs <- read.table("2 other eggs")
```

A fenti esetben CSAK ÉS KIZÁRÓLAG akkor dobjuk ki a tojást, vagy a lisztet, ha az zsizsikes, vagy záp.

Általános esetben:

```
if (logikai kifejezés) feltételesen futtatandó kód sor
```

Több sornyi kódot is futtathatunk, amennyiben azt kód blokkban helyezzük el.

```
if (logikai kifejezés){
  1. feltételesen futtatandó kód sor
  2. feltételesen futtatandó kód sor
  3. feltételesen futtatandó kód sor
  4. feltételesen futtatandó kód sor
}
```

Ez esetben, ha a logikai kifejezés igaz, vagyis a feltételünk teljesül, a kód blokk mind a 4 sora lefut, ha nem, akkor egyik sem, és közvetlenül a blokk után jövő utasítás hajtódik végre. A dagasztós példánál maradva: ha sem nem zápok a tojások, sem nem zsizsikes a liszt, akkor rögtön elkezdjük gyúrni a tésztát.

```
x <- 4
if (x>3) print("x is greater than 3")
```

```
## [1] "x is greater than 3"
```

```
if (x<3) print("x is less than 3")
if (x==4) print("x equals 4")
```

```
## [1] "x equals 4"
```

Gyakran megesik, hogy ha a vizsgálat negatív eredménnyel zárul, akkor nem elegendő, ha nem fut le a kikötés után a kód, hanem ez esetben mást szeretnénk futtatni.

Példának okáért írjunk egy olyan kódot, ami beolvas egy fájlt, majd annak egy oszlopából szétválogatja a kettővel osztható, és a kettővel nem osztható számokat. A teljes példát csak a ciklusok bemutatása után tudjuk megoldani, de az alapokat már most lefektetjük.

Ehhez első sorban a modulo (%%) aritmetikai operátorra lesz szükségünk.

A modulo operátor a bal oldalát a jobb oldalával, majd megadja az osztás maradékát.

például:

```
4 %% 3
```

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

Kifejtve: 4 / 3 = 1, marad az 1

```
4 %% 2
```

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

4 / 2 = 2, marad a 0

Vagyis, ha azt szeretnénk, hogy egy kód csak akkor fusson le, ha a vizsgált szám kettővel osztható:

```
x <- 4
if (x%%2==0) print("x is divisible by 2") else print("it's an odd number")
```

```
## [1] "x is divisible by 2"
```

```
x <- 15
if (x%%2==0) print("x is divisible by 2") else print("it's an odd number")
```

```
## [1] "it's an odd number"
```

Bekerült az “else” a képbe. Ha a kitétel igaz, csak a `print("x is divisible by 2")` ha a kitétel hamis, rögtön az `else` utáni részre ugrik és csak a `print("it's an odd number")` fut le.

Megesik, hogy ez nekünk nem elég. Több felé kell szortíroznunk az információt:

```
if (x%%2==0) print ("x is divisible by 2") else if (x%%3==0)
  print("x is divisible by 3") else if(x%%5==0)
    print("x is divisible by 5") else print("where did this come from")
```

```
## [1] "x is divisible by 3"
```

Kód blokkot használva kicsit átláthatóbbá tehetjük a dolgot:

```
if (x%%2==0) {
  print ("x is divisible by 2")
} else if (x%%3==0) {
  print("x is divisible by 3")
} else if(x%%5==0) {
    print("x is divisible by 5")
} else {
    print("where did this come from?")
}
```

```
## [1] "x is divisible by 3"
```

```
print("so we shall next")
```

```
## [1] "so we shall next"
```

Vagyis, ha x osztható kettővel lefut az első print, majd a kikötések utáni sor fut le, vagyis az eredmény:

```
"x is divisible by 2"
"so we shall next"
```

Ha x == 15

```
"x is divisible by 5"
"so we shall next"
```

Ezt egyébként if – else blokknak hívják.

Figyeljük meg, hogy az if – else blokk futtatása azonnal megáll, amint valamelyik feltétel if, vagy else if feltétel teljesül, rögtön a blokk utáni első sorra ugrik a végrehatjás.

Mint már említettük ezek azok a nyelvi konstrukciók, melyek lehetővé teszik az ismétlődő munkavégzést. Iparban a tartályok és forgó berendezések teszik ezt lehetővé. Főzés esetén pedig a tál és a fakanál, vagy épp a kezünk.

Újra, és újra elvégezzük ugyanazt a műveletet, amíg célhoz nem érünk.

Próbáljuk megírni a kedvenc sum függvényünket függvényhívás nélkül!

```
x <- c(5,10,15,20,25)
sum <- 0
for (value in x){
    sum <- sum + value
}
```

Lefuttatva az Rstudio jobb fölső moduljában, a változók között láthatjuk, hogy a sum 75-tel lett egyenlő. Megoldásunk eredményét egy szokásos sum(x) hívással ellenőrízhetjük.

Vagyis a value változó minden iterációkor közvetlenül felveszi x soron következő értéket:

1. iteráció `value = x[1] = 5`

```
sum <- sum + value
sum <- 0 + 5
```

2. iteráció: `value = x[2] = 10`

```
sum <- sum + 10
sum <- 5 + 10
```

3. iteráció: `value = x[3] = 15`

```
sum <- sum + 15
sum <- 15 + 15
```

4. iteráció: `value = x[4] = 20`

```
sum <- sum + 20
sum <- 30 + 20
```

5. iteráció: `value = x[5] = 25`

```
sum <- sum + 25
```

Figyeljük meg, hogy nem `i`*-nek hívjuk a változót, hanem value-nak. Hasznos, ha olyan nevet adunk a változóinknak, ami alapján egyből tudjuk, mivel is van dolgunk. Az*`i`,`j`,`k`,`l` *betűket az egymás utáni dimenziójú indexváltozókra szokás használni, ezekről többet a kiadott loops.pdf-ben olvashattok. Azonban mivel itt a value nem az indexváltozó, hanem közvetlenül x i-edik értéke így illik legalább* `value`*-nak hívni, de inkább valami adjunk neki valami beszéldes nevet, ami alapján egyértelmű, hogy milyen értékkel dolgozunk épp a ciklusban. Ha például egy* `data.frame` *megfelelő oszlopán futunk végig:*

```
for(life.span in experiment$life.span){
    ...
}
```

Ezek segítségével már megírhatjuk a szortírozós kódunkat, amit a vezérlési utasításoknál félbehagytunk:

```
if (x%%2==0) {
  print ("x is divisible by 2")
} else if (x%%3==0) {
  print("x is divisible by 3")
} else if(x%%5==0) {
    print("x is divisible by 5")
} else {
    print("where did this come from")
}
```

Ha már x volt a változó, maradjunk annál, de kérek mindenkit, erősen ráncolja a homlokát, és jegyezze meg magában, hogy "ez bizony így nagyon nem szép!"

```
vector <- 1:100
div.2 <- NULL
div.3 <- NULL
div.5 <- NULL
others <- NULL

for (x in vector){
  if (x%%2==0) {
    div.2 <- c(div.2, x)
  } else if (x%%3==0) {
    div.3 <- c(div.3, x)
  } else if(x%%5==0) {
    div.5 <- c(div.5, x)
  } else {
    others <- c(others, x)
  }
}

div.2
```

```
##  [1]   2   4   6   8  10  12  14  16  18  20  22  24  26  28  30  32  34
## [18]  36  38  40  42  44  46  48  50  52  54  56  58  60  62  64  66  68
## [35]  70  72  74  76  78  80  82  84  86  88  90  92  94  96  98 100
```

```
div.3
```

```
##  [1]  3  9 15 21 27 33 39 45 51 57 63 69 75 81 87 93 99
```

```
div.5
```

```
## [1]  5 25 35 55 65 85 95
```

```
others
```

```
##  [1]  1  7 11 13 17 19 23 29 31 37 41 43 47 49 53 59 61 67 71 73 77 79 83
## [24] 89 91 97
```

Ha megnézzük, a hárommal osztható páros számok, csak a div.2-be kerültek be, ahogyan a 10-el osztható számok is, míg a 15-el oszthatók csak a div.3-ba.

Ez azért van mert csak egyszer az `if-else` blokk-on belül csak az egyik utasítás futhat le. Ha az `if`, vagy valamelyik `else if` utáni feltétel teljesül, lefut az utána lévő kód, majd az `if-else` blokk végére ugrik az interpreter.

Ezt úgy tudjuk elkerülni, ha nem használunk if-else blokkot, hanem mindegyik feltételt `if`-fel kezdjük.

```
vector <- 1:100
div.2 <- NULL
div.3 <- NULL
div.5 <- NULL
others <- NULL

for (x in vector){
  if (x%%2==0) {
    div.2 <- c(div.2, x)
  }
  if (x%%3==0) {
    div.3 <- c(div.3, x)
  }
  if(x%%5==0) {
    div.5 <- c(div.5, x)
  } else {
    others <- c(others, x)
  }
}

div.2
```

```
##  [1]   2   4   6   8  10  12  14  16  18  20  22  24  26  28  30  32  34
## [18]  36  38  40  42  44  46  48  50  52  54  56  58  60  62  64  66  68
## [35]  70  72  74  76  78  80  82  84  86  88  90  92  94  96  98 100
```

```
div.3
```

```
##  [1]  3  6  9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57 60 63 66 69
## [24] 72 75 78 81 84 87 90 93 96 99
```

```
div.5
```

```
##  [1]   5  10  15  20  25  30  35  40  45  50  55  60  65  70  75  80  85
## [18]  90  95 100
```

```
others
```

```
##  [1]  1  2  3  4  6  7  8  9 11 12 13 14 16 17 18 19 21 22 23 24 26 27 28
## [24] 29 31 32 33 34 36 37 38 39 41 42 43 44 46 47 48 49 51 52 53 54 56 57
## [47] 58 59 61 62 63 64 66 67 68 69 71 72 73 74 76 77 78 79 81 82 83 84 86
## [70] 87 88 89 91 92 93 94 96 97 98 99
```