Functions

Eszter Ari, Tamas Kadlecsik

March 16, 2017

Házi feladat megoldás

1. Feladat

```
ex1 <- read.table("experiments/experiment1.csv", header=T, sep=",", comment.char="")
lifespan <- ex1$died - ex1$born
ex1$lifespan <- lifespan
head(ex1)
```

```
##   Specimen.ID born died treatment    diet weight lifespan
## 1      #91146  312 1464   treated control     21     1152
## 2      #94706  835 2079   control control     24     1244
## 3      #90078  685 1869   control treated     20     1184
## 4      #92785  210 1373   control control     23     1163
## 5      #97457   55 1288   control treated     17     1233
## 6      #98858  495 1702   control treated     15     1207
```

```
write.table(ex1, "result1.csv", sep=",", row.names=F)
```

2. Feladat:

```
orig_names <- paste("experiments/experiment", 1:10, ".csv", sep="")
orig_names
```

```
##  [1] "experiments/experiment1.csv"  "experiments/experiment2.csv"
##  [3] "experiments/experiment3.csv"  "experiments/experiment4.csv"
##  [5] "experiments/experiment5.csv"  "experiments/experiment6.csv"
##  [7] "experiments/experiment7.csv"  "experiments/experiment8.csv"
##  [9] "experiments/experiment9.csv"  "experiments/experiment10.csv"
```

```
dir.create("results", showWarnings=F)
new_names <- gsub(".csv","_mod.csv", orig_names)
new_names <- gsub("experiments/","results/", orig_names)
new_names
```

```
##  [1] "results/experiment1.csv"  "results/experiment2.csv"
##  [3] "results/experiment3.csv"  "results/experiment4.csv"
##  [5] "results/experiment5.csv"  "results/experiment6.csv"
##  [7] "results/experiment7.csv"  "results/experiment8.csv"
##  [9] "results/experiment9.csv"  "results/experiment10.csv"
```

```
average_cc <- matrix(0, ncol=2, nrow=10)  # treatment: control diet: control
colnames(average_cc) <- c("average_weight", "average_lifespan")
average_cc
```

```
##       average_weight average_lifespan
```

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

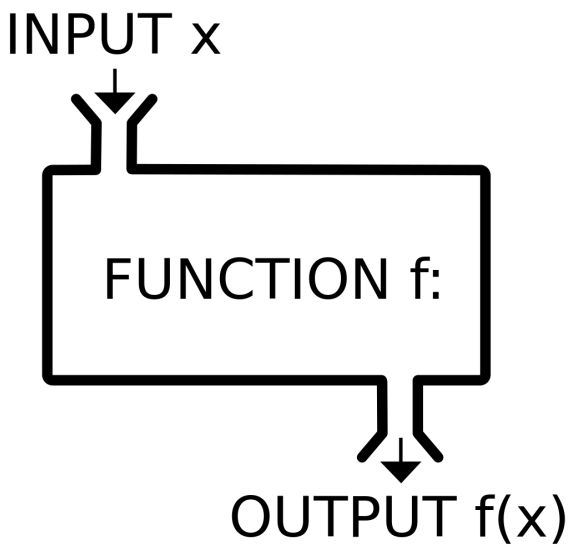
```
average_tt <- matrix(0, ncol=2, nrow=10)  # treatment: treated diet: treated
colnames(average_tt) <- c("average_weight", "average_lifespan")
average_ct <- matrix(0, ncol=2, nrow=10)  # treatment: control diet: treated
colnames(average_ct) <- c("average_weight", "average_lifespan")
average_tc <- matrix(0, ncol=2, nrow=10) # treatment: treated diet: control
colnames(average_tc) <- c("average_weight", "average_lifespan")

for (i in 1:length(orig_names)) {
  # write result tables includes lifespan
  Tab <- read.table(orig_names[i], header=T, sep=",", comment.char="")
  lifespan <- Tab$died - Tab$born
  Tab$lifespan <- lifespan
  write.table(Tab, new_names[i], sep=",", row.names=F)

  # calculate averages
  # treatment: control diet: control
  cc <- Tab[Tab$treatment=="control" & Tab$diet=="control",]
  average_cc[i,1] <- mean(cc$weight)
  average_cc[i,2] <- mean(cc$lifespan)

  # treatment: treated diet: treated
  tt <- Tab[Tab$treatment=="treated" & Tab$diet=="treated",]
  average_tt[i,1] <- mean(tt$weight)
  average_tt[i,2] <- mean(tt$lifespan)

  # treatment: control diet: treated
  ct <- Tab[Tab$treatment=="control" & Tab$diet=="treated",]
  average_ct[i,1] <- mean(ct$weight)
  average_ct[i,2] <- mean(ct$lifespan)

  # treatment: treated diet: control
  tc <- Tab[Tab$treatment=="treated" & Tab$diet=="control",]
  average_tc[i,1] <- mean(tc$weight)
  average_tc[i,2] <- mean(tc$lifespan)
}

all_averages <- list(average_cc, average_ct, average_tc, average_tt)
all_averages[[1]]
```

```
##       average_weight average_lifespan
##  [1,]       19.88406         1199.565
##  [2,]       19.71250         1201.225
##  [3,]       19.83333         1197.318
##  [4,]       19.38235         1200.147
```

```
##  [5,]       19.18966         1202.293
##  [6,]       19.37879         1195.879
##  [7,]       19.67241         1205.517
##  [8,]       19.47143         1196.329
##  [9,]       19.77778         1198.444
## [10,]       19.59091         1199.667
```

Matematika

Kedzjük a matekkal. Ha megnézzük a legtöbb matek könyvet, vagy a wikipediát, a függvény definíciója általában valami borzasztó bonyolult szöveg, amit igen egyszerűen összefoglal az alábbi ábra:



Van valami inputunk, azt megkapja a függvény, csinál vele valamit, majd egy módosított értéket ad vissza:

$$f(x) = x + 1$$

Jelen függvényünk nemes egyszerűséggel minden számra visszaadja az adott szám egyel megnövelt értékét.

x a függvény **paramétere**.

Amennyiben a függvény végre kívánjuk hajtani egy objektumon (számon), annak matematikai jelölése a következő:

$$f(5)$$

A függvényünk megkapta az ötöt, vagyis most a függvényhívás **argumentuma**: 5 (röviden: a függvény argumentum: 5. Az argumentum bekerül a függvény hasába:

$$f(5) = 5 + 1$$
$$f(5) = 6$$

Az

$$f(x) = x + 1$$

kifejezést **függvénydefiníciónak** hívjuk.

Egy függvénynek lehet több paramétere is:

$$g(x, y) = x + y$$
$$g(3, 5) = 8$$
$$h(x, y) = x * y - 15$$
$$h(3, 5) = 0$$

Egy függvény felhasználhat más függvényket is:

$$i(x, y, z, a, b) = \frac{f(x) + g(y, z)}{h(a, b)}$$
$$i(7, 9, 8, 7, 5) = \frac{f(7) + g(9, 8)}{h(7, 5)}$$
$$i(7, 9, 8, 7, 5) = 1.25$$

Megoldható, hogy bizonyos feltételek teljesülése esetén másképp viselkedjen a függvény:

$$j(x, y) = \begin{cases} x + 1, & x < 1 \\ y^2, & 1 \leq x < 100 \\ x^y, & 100 \leq x \end{cases}$$

Jelen függvényünk $x + y$-t ad vissza, ha a beadott x egynél kisebb. Ha 1 és 100 között van, akkor y négyzetét adja vissza. A harmadik esetben pedig x az y-adikont adunk vissza. Az, hogy a 2. esetben x-et ne használjuk fel teljesen megengedett. Sőt, mindhárom esetben vizsgálhatnánk x-et, és csak y-al kezdenénk valamit. Tehát:

$$j(0.5, 6) = 1.5$$

x kisebb egy, így $j(x, y) = x + 1$

$$j(15, 2) = 4$$

x 1 és 100 között van, így y négyzetét adjuk vissza, vagyis $j(x, y) = y^2$

$$j(2, 100) = 2^{100}$$

x nagyobb vagy egyenlő 100-al, x y-adik hatványát adjuk vissza, vagyis $j(x, y) = x^y$

R

Nos, lássuk hogy megy a függvénydefiníció R-ben:

```
function(param1, param2, ..., paramN){
  1. utasítás
  2. utasítás
  3. utasítás
}
```

Látható, hogy a szintaktika erősen a matekból jön. A kód blokk kapcsoszárójele, a matematikai függvények feltételes definíciójára hajaz.

Azonban ezzel még nem vagyunk kész!

Függvényeink ugyanolyan objektumok, mint az általunk létrehozott `data.frame`-ek, így ezeket is változóhoz kell rendelnünk, hogy hivatkozni tudjunk rájuk. A matematikai függvényeinket közvetlenül neveztük el `f`, `g`, `h`, `i`, `j`-nek. R-ben azonban a függvénydeklaráció és az elnevezés kettéválik. (Java-ban, C-ben nincs function kulcsszó, ott a függvénydefiníció jobban hasonlít a matematikaira). Tehát az általános függvénydefiníció:

```
function.name <- function(param1, param2, ..., paramN){
  1. utasítás
  2. utasítás
  3. utasítás
}
```

Definiáljuk hát első függvényünket!

```
my.square <- function(x){
  x^2
}
```

Jelen függvényünk négyzetre emeli az argumentumként beadott értéket.

```
a <- 1:5
my.square(a)
```

```
## [1]  1  4  9 16 25
```

```
b <- my.square(a)
print(b)
```

```
## [1]  1  4  9 16 25
```

Nézzük meg mi történik, ha egy függvényen belül több utasítást is kiadunk!

```
my.erroneous.square.and.sum <- function(x){
  x^2
  sum(x)
}
my.erroneous.square.and.sum(a)
```

```
## [1] 15
```

```
b <- my.erroneous.square.and.sum(a)
b
```

```
## [1] 15
```

```
sum(a)
```

```
## [1] 15
```

Látható, hogy csupán az utolsó utasítás értékét kaptuk vissza! Próbálgassuk tovább!

```
my.still.erroneous.square.and.sum <- function(x){
  x^2
  result <- sum(x)
}
```

```
result
# Error: object 'result' not found
```

```
b <- my.still.erroneous.square.and.sum(a)
b
```

```
## [1] 15
```

Több dolgot és észrevehetünk egyszerre:

- A függvény utolsó kifejezésének értéke lesz a függvény értéke
- A függvényen belül definiált változókat kívülről nem érhetjük el.

(Ezen lehet érdemes egy kicsit elmerengeni, vagy többször végigfutni)

Az első sort később majd még pontosítjuk, a másodikkal foglalkozzunk egy kicsit:

Minden függvény, **környezetet** vagy **scope**-ot definiál. Vagyis a függvény látja a rajta kívül lévő változókat, de kívülről nem látjuk a függvényen belülieket. Elsőre zavarónak tűnhet, de ez nagyban segíti a moduláris, újrafelhasználható kódok írását. Képzeljük csak el például, milyen idegesítő lenne, ha a beépített függvények összes változóját látnánk kívülről. Egy csomó változónevet nem használhatnánk, nehogy felülírjuk a `sum`, `mean`, `read.table`, vagy a `paste` belső változóit! Ráadásul az RStudio Environment ablaka is teljesen használhatatlan lenne, hiszen tele lenne midnennel szemetelve.

Apropó környezet:

A legkülső környezet, ahol eddig dolgoztunk a globális környezet. Az itt létrehozott változókat minden függvény látja. Az Rstudio jobb fölső moduljában, ha megnézzük "Global Environment"-nek hívja a környezetet, ahol a változókat kilistázza.

Mint már a matematikai példánkon is láttuk a függvények akkor jönnek jól, ha van egy gyakran előforduló, jól definiálható probléma amelyet szeretnénk megoldani. Ilyenkor ezt függvénybe csomagoljuk. Az előző részben felhozott példánál maradva, az

$$i(7, 9, 8, 7, 5)$$

Hívás jóval rövidebb, gyorsabban legépelhető, és könnyebben fejben tartható, mint annak teljes kifejtése:

$$\frac{7 + 1 + 9 \times 8}{7 * 5 - 15}$$

Gyakorlatilag adott problémára történő megfelelő függvények írása az alapszintű programozás esszenciája. Úgyis mondhatnánk: az első lépés a programozóvá válás rögös útján, az első újrafelhasználható függvény.

`return()`

Igen gyakran megesik, hogy nekünk nem a legutolsó sorra van szükségünk egy függvényből, hanem bizonyos feltételek esetén más és más értéket szeretnénk vissza adni. Erre szolgál a return utasítás.

```
my.test.function <- function(x){
  my.square <- x^2
  sum <- sum(x)
  return(my.square)
```

```
}

my.test.function(a)
```

```
## [1]  1  4  9 16 25
```

Vagyis mi az első sor eredményét szerettük volna kihozni a függvényből, így a `return` utasításba azt tettük. Tehát a függvény értéke mindig a lefutott `return` utasítással lesz egyenlő.

Jó prgramozói szokás midig explicit módon jelölni a visszatérési értéket, ezért lehetőleg ne hagyatkozzunk arra, hogy az "utolsó sor majdcsak visszatér"! Csupán azért mutattuk be a `return` nélküli függvény működést, mert sajnos nem kevés ilyen kóddal lehet találkozni.

Vegyük észre, hogy nyugodtan felülírhattam a `my.square` változót a függvényen belül, mivel a függvény belső változói kívülről nem látszanak, a globális `my.square` függvényünk sértetlen maradt

```
my.square(2)
```

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

`print()` vs `return()`

R-ban a statisztikai függvények igen gyakran printelnek a konzolra eredményt megfelelően katyvaszos formában, aminek tagjait később el tudjuk érni, ha a függvény értékét változóhoz rendeljük. Ezt a `print()` utasítással érik el:

```
my.complex.function <- function(x) {
  square <- x^2
  sum <- sum(x)
  print(paste("The result of my complex function is:", sum, sep=" "))
  return(sum)
}

b <- my.complex.function(a)
```

```
## [1] "The result of my complex function is: 15"
```

```
b
```

```
## [1] 15
```

Vagyis a függvény futása közben lefutott a `print` utasítás. Kiszólt nekünk valamit, majd futott tovább, elérte a `return` utasítást, és a függvény felvette a `sum` változó értékét. A `print()`-tel történő kiszólás jól jöhet debuggoláskor. Mivel a függvényünk változóit közvetlenül nem tudjuk elérni, a függvény futása közben a `print()` utasítással tudjuk őket megtekinteni.

Natív vs Interpretált

Az eddig leírtak az R-ben írt függvényekre vonatkoztak. Ha beírjuk az eddig írt függvényeink nevét, függvényhívó operátor nélkül a konzolba, megkapjuk a függvény kódját:

```
my.test.function
```

```
## function(x){
##   my.square <- x^2
##   sum <- sum(x)
##   return(my.square)
```

```
## }
```

Ez a függvény egy interpretált függvény, mivel a parancsértelmező értelmezi és futtatja.

Amennyiben úgynevezett natív (beépített) függvények nevét írjuk be, mást kapunk:

```
str
```

```
## function (object, ...)
## UseMethod("str")
## <bytecode: 0x1f83318>
## <environment: namespace:utils>
```

Az `str()` függvény gépikódra lefordult, vagyis csupán az argumentumot kell a parancsértelmezőnek feldolgoznia, a függvény utasításait a számítógép közvetlenül tudja futtatni.

Több visszatérési érték

Van, hogy több változót szeretnénk kihozni egy függvényből

```
my.multiple.return.values  <- function(x){
  result <- x^2
  sum <- sum(x)
  return(list(res=result, sum=sum))
}
```

Jelen esetben szeretnék visszaadni mind a `result`, mind a `sum` változó értékét.

```
my.multiple.return.values(a)
```

```
## $res
## [1]  1  4  9 16 25
##
## $sum
## [1] 15
```

```
b <- my.multiple.return.values(a)
b
```

```
## $res
## [1]  1  4  9 16 25
##
## $sum
## [1] 15
```

Ha több külböző változót szeretnénk visszadni, azt listába csomagolva tudjuk megtenni!

Kezdeti értékek

A `read.table()`-nél láttuk, a következő jelölést:

```
read.table(file, header = FALSE, sep = "", ...)
```

A `read.table()` kódja úgy van megírva, hogy amennyiben a `header`, és a `sep` értékét nem állítjuk be, akkor alapesetben kapnak egy kezdőértéket, ami a header esetében `FALSE`, a sep esetében `""`.

Írjunk saját kezdeti értékkel ellátott függvényt!

```
my.initial.value <- function(x, y=2){
  return(x*y)
}
```

Amennyiben y-nak nem adunk meg semmit, értéke automatikusan 2 lesz.

```
my.initial.value(5)
```

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

De

```
my.initial.value(5,7)
```

```
## [1] 35
```

```
my.initial.value()
# Error in my.initial.value() : argument "x" is missing, with no default
my.initial.value(5,7,9)
# Error in my.initial.value(5, 7, 9) : unused argument (9)
```

Láthatjuk, hogy ha mindkét paraméternek értéket adunk, egyszerűen lefut a függvény, és a két argumentumot összeszorozza.

Ha csak `x` kap értéket, `y` kezdeti értékét helyettesíti be (2), és azzal szorozza be a kapott argumentumot.

Ha se `x`, se `y` értékét nem adjuk meg. Ennek hatására hibát kapunk, minek következtében nem fut le a kód. Függvényhívásnál kezdeti értékkel nem rendelkező paramétereknek mindig értéket kell adnunk!

Végezetül, ha túl sok argumentumot adunk, szintén hibát kapunk.

Elnevezett argumentumok

Itt jön jól a paraméter – argumentum megkülönböztetés. Ugyanis mint láthattuk, a paramétereinknek mindig van nevük. Lehet ez `x`, vagy `file`, `header`, akármi, de nevük biztos, hogy van. Ezzel szemben kétféle képpen adtunk át függvénynek argumentumot. Vagy pozíció alapján, vagy névvel.

Ha pozíció alapján akkor egyszerűen az első argumentum, az első paraméter helyére lesz behelyettesítve:

```
my.initial.value(12,4)
```

```
## [1] 48
```

`x 12`, míg `y 4` lett.

Azonban a `read.table()`-nél láttuk a következő hívást

```
some.data <- read.table("path/to/file", header = T, sep=',')
```

Itt a `file` argumentum pozíció alapján került be, míg a `header`, és a `sep` névvel.

Előző függvényünket meghívhattuk volna így is:

```
my.initial.value(y=4, x=12)
```

```
## [1] 48
```

Vagyis, ha az adott paraméterre a nevével hivatkozunk, teljesen lényegtelen, hogy mi a pozíciója.

Általában úgy illik függvényt hívni, hogy a pozíció alapján átadott paraméterekkel kezdünk, majd utána írjuk a névvel átadottakat. Ugyan az R megengedi ezek keverését, azonban névvel történő átadás után borul a sorrend, és nehéz követni, hogy éppen melyik paraméter pozíciójában vagyunk.

Pár függvény, ami eddig hasznos lett volna

Az előző órán írtunk egy scriptet, ami bemutatta a `sum()` függvény működését.

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

Azonban ennek működése nem túl szerencsés. Képzeljük el, hogy a `summa` változót még nagyon régen definiáltuk. Megírtuk ezt a `for` ciklust, majd később tovább alakítjuk. Egy kusza, sok `if-else` kitétellel, elágazásokkal teli kódunk született. Ezen kódban a `summa` változó sorsát végigkövetni igen csak nehézkes. Amennyiben vannak helyzetek, ahol a `summa` értékét vizsgáljuk meg, azonban a kódban valahol már megváltoztattuk az értékét, simán lehet, hogy nem a várt eredményt kapjuk, vagyis bugos a kód. Nem elhasal, csak rosszul működik. Ezt kidebuggolni rémálom lenne!

Ezért érdemes úgy megírni a függvényeket, hogya külvilágról csak a paramétereiken keresztül értesüljenek, és csak belső változókat használjanak.

Mivel kívülről nem látják a függvény változóit, és paramétereinek az értékét, nem fordulhat elő, hogy egy belső változót a kódnak egy idegen része módosít, vagy felülír. Sokkal modulárisabb a kód, kisebb szeletét kell átnyálazni a probléma feltárásához.

Nézzük ezt függvényként:

```
x <- c(5,10,15,20,25)
my.redefined.sum <- function(vector){
  container <- 0
  for (value in vector){
        container <- container + value
  }
  return(container)
}
```

Jelen esetben nem áll fenn a veszélye, hogy a kódunk egy részében fölülírjuk a `container` változónkat, hiszen csak a függvényen belül elérhető.

Burkolók

A burkoló (wrapper) függvények, olyan függvények, amik egy általános használatra írt függvényt rögzítenek olyan állapotra, amit mi általában használunk.

```
my.read.csv <- function(file, header=TRUE, sep=",", quote="", comment.char=""){
  return(read.table(file=file, header=header, sep=sep, quote=quote,
                    comment.char=comment.char))
}
```

Mivel általában .csv-t olvasunk be, a `header`, `sep`, `quote`, `comment.char` értekeket szinte mindig ugyanarra állítjuk. Ezt igen melós minden beolvasáskor megtenni, ezért érdemes írni egy burkolót, ami elvégzi helyettünk. A fenti függvény nem csinál mást, mint általunk gyakran használt értékekkel meghívja, majd visszaadja a `read.table()`-t. Így a `','`-s filebeolvasás leegyszerűsödik ennyire:

```
some.data <- my.read.csv('path/to/file')
```

A `';'`-s pedig:

```
some.data <- my.read.csv('path/to/file', sep=";")
```

Ilyen burkolók egyébként be vannak építve az R-be, majdnem ugyanilyen formában:

```
read.csv
```

```
## function (file, header = TRUE, sep = ",", quote = "\"", dec = ".",
##     fill = TRUE, comment.char = "", ...)
## read.table(file = file, header = header, sep = sep, quote = quote,
##     dec = dec, fill = fill, comment.char = comment.char, ...)
## <bytecode: 0x3dd22c0>
## <environment: namespace:utils>
```

```
read.csv2
```

```
## function (file, header = TRUE, sep = ";", quote = "\"", dec = ",",
##     fill = TRUE, comment.char = "", ...)
## read.table(file = file, header = header, sep = sep, quote = quote,
##     dec = dec, fill = fill, comment.char = comment.char, ...)
## <bytecode: 0x3e657f0>
## <environment: namespace:utils>
```

```
read.delim
```

```
## function (file, header = TRUE, sep = "\t", quote = "\"", dec = ".",
##     fill = TRUE, comment.char = "", ...)
## read.table(file = file, header = header, sep = sep, quote = quote,
##     dec = dec, fill = fill, comment.char = comment.char, ...)
## <bytecode: 0x3ed81b0>
## <environment: namespace:utils>
```

```
read.delim2
```

```
## function (file, header = TRUE, sep = "\t", quote = "\"", dec = ",",
##     fill = TRUE, comment.char = "", ...)
## read.table(file = file, header = header, sep = sep, quote = quote,
##     dec = dec, fill = fill, comment.char = comment.char, ...)
## <bytecode: 0x3f682f0>
## <environment: namespace:utils>
```

A különbség annyi, hogy az "utolsó sor visszatér" működést használja ki, illetve, hogy van mind a `read.csv()` paraméterlistájának, mind a megívhott `read.table()` argumentum listájának végén látunk egy `...`-ot. (*Angolul ezt ellipsis-nek hívják, ha valaki utánanézne*). Ez semmi mást nem jelent, mint hogy a felsorolt paramétereken túl, rengeteg argumentumot lehet átadni a `read.csv`-nek, az pedig ezeket váltoatlan formában átadja a `read.table()`-nek. A `read.table()`-nek rengeteg további állítható paramétere van, melós és fölösleges lenne midet fölsorolni a `read.csv()`-ben.

`source(file)`

Ha már megírtuk a kíváló `my.read.csv` függvényünket, lehet, hgoy még több ilyet írunk. Ezek bizonyos része bármilyen kódnál jól jöhet, míg mások csak az adott projectben, míg megint mások csak az adott probléma megoldása során. Amikor több száz függvényünk vagy egy fájlban, nehéz eldönteni, hogy melyik melyik kategóriába tartozik, így érdemes ezeket külön fájlokban tárolni. Ahhoz azonban, hogy egy másik script fájlban tárolt függvényt az adott scriptünkben le tudjunk futtatni importálnunk kell azt a file-t, amiben a scriptet tároltuk a `source("filename.r")` függvénnyel.

Feladat

Egyszerűsítsétek a múltkori 2. házi feladat megoldását saját függvény(ek) írásával és használatával!