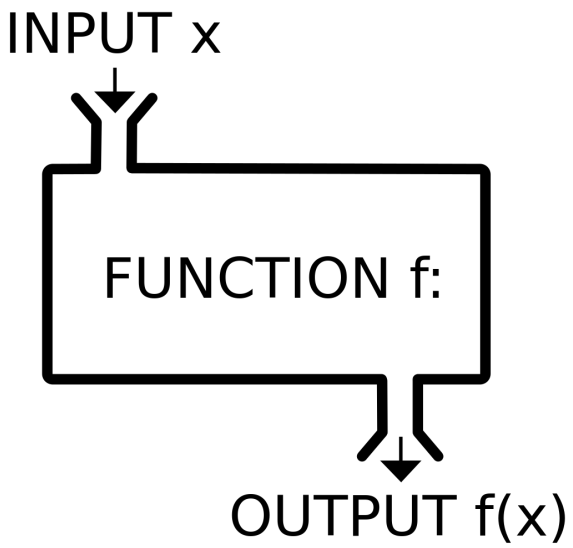
Functions

Tamas Kadlecsik

March 5, 2016

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, meg 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!

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

Jelen függvényünk négyzetre emeli az argumentumként beadott értéket.

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

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

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

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

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

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

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

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

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

```
sum(a)
```

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

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

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

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

```
b <- 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, raed.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 mindenki 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.

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

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 jelölni a visszatérési értéket, ezért lehetőleg ne hagyatkozzunk az "utolsó sor visszatér" működésre! Csupán azért mutattuk be a return nélküli függvény működést, mert sajnos nem kevés ilyen kóddal lehet találkozni.

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

```
square(2)
```

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

print() vs return()

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:

```
mischievous.function <- function(x) {
  square <- x^2
  my.sum <- sum(x)
  print(my.sum)
  return(square)
}

b <- mischievous.function(a)
```

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

```
b
```

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

Vagyis a függvény futása közben lefutott a `print` utasítás. Kiszólt nekünk valamit, jelen esetben a `my.sum` értékét, majd futott tovább, elérte a `return` utasítást, és a függvény felvette a square változó értékét. A print()-el 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:

```
test.function
```

```
## function(x){
##   square <- x^2
##   my.sum <- sum(x)
##   return(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: 0x26731d8>
## <environment: namespace:utils>
```

Az str() függvény gépikódra lefordult, vagyis csupán az argumentumot kell a parancsértelmezőnek feldolgoznia, a függvény utasításait a számítógép közvetlenül tudja futtatni. Működése nagyban hasonlít az R-ben megírtakéhoz, azonban R-ben a függvények annyiban térnek el az általánostól, hogy más nyelvekben a függvényeknek nincs kötelező visszatérési értékük. Így a str()-nek sincs, csupán a konzolra printeli, az adott objektum struktúráját. Ezért nem tudtuk változóhoz rendelni, és kiírni az első házi feladatban.

Több visszatérési érték

Van, hogy több változót szeretnénk kihozni egy függvényből

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

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

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

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

```
b <- 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!

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

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

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

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

De

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

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

```
initial.value()
# Error in initial.value() : argument "x" is missing, with no default
initial.value(5,7,9)
# Error in 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 két féle képpen adtunk át függvénynek argumentumot. Vagy pozíció alapján, vagy névvel.

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

```
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:

```
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 handoutban írtunk egy scriptet, ami bemutatta a sum függvény működését.

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

Azonban ennek működése nem túl szerencsés. Képzeljük el, hogy a sum 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 sum változó sorsát végigkövetni igen csak nehézkes. Amennyiben vannak helyzetek, ahol a sum 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)
redefined.sum <- function(vector){
  container <- 0
  for (value in vector){
        container <- container + value
  }
  return(container)
}
```

Jelen esetben nem áll fenn a veszélye, hogy bárki fölülírja 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.

```
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-tÍgy a `','`-s filebeolvasás leegyszerűsödik ennyire:

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

A `';'`-s pedig:

```
some.data <- 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: 0x2864118>
## <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: 0x19a9fb0>
## <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: 0x28a4130>
## <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: 0x1b208a0>
## <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, végtelen argumentumot lehet átadni a read.csv-nek, az pedig ezeket változatlan 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.

Létezik write.csv is, de az például nem ennyire kényelmes, így érdemes sajátot használni helyette

```
write.csv.custom <- function(x, file, quote=F, sep=',', row.names=F, ...) {
  write.table(x, file, quote=quote, sep=sep, row.names=row.names, ...)
}
```

Az eredetiben például a a row.names-t true-ra állítja, aminek általában nincs sok értelme. Azért a végére tettük a custom jelölést, mert így ha elfelejtjük mivel különböztettük meg az eredetitől csak elkezdjük írni, hogy `write.csv` és a felajánlott listából már eszünkbe juthat, melyik a miénk. Fölül pedig azért nem írtuk, mert ha dolgozunk egy projecten, simán lehet, hogy még a befejezése előtt máshova sodor minket az élet, és másnak kell folytatnia. Ilyen esetben az illető elvárná, hogy a `write.csv` úgy működjön mint amit a `?write.csv` állít, de nem fog, mert mi fölülírtuk. Röviden: beépített függvényeket fülüírni bunkó dolog.

Ez a függvény igen általános, és akár életünk végig ki is szolgálhat minket. Azonban a 100 kísérlet feldolgozásakor igen sokszor kellett a paste() függvényt használnunk, hogy létrehozzuk a fájlneveinket, azt is érdemes lett volna beburkolni:

```
write.result <- function(x, path, filename, num='') {
  path <- paste(path, filename, sep='')
  destination <- paste(path, '.csv', sep=as.character(num))
  write.csv.custom(x, destination)
}
```

Arra, hogy mi az az utoasítás, amit még érdemes a függvénybe tenni, és mi az amit már nem, csak sok sok tapasztalat és refaktorálás árán tud rájönni az ember.

source(file)

Ha már megírtuk a kíváló `write.csv.custom` 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.

Tételezzük fel, hogy van két fájlunk, `functions.R` és `chinese_experiment_processor.R`

A `chinese_experiment_processor.R` olvassa be a 100 fájlt, kiértékeli, és írja ki. De régóta magunkkal hordozott, bővített és tökéletesített `functions.R`-fájlunkban van definiálva a `write.result` függvényünk, amit mi szeretnénk felhasználni. Ehhez az kell, hogy az `functions.R` lefusson, és definiálja a függvényünket még mielőtt a `chinese_experiment_processor.R` felhasználná. Ezt pedig a `source(file)` függvénnyel tudjuk megtenni.

Tegyük fel, hogy a `function.R` a `chinese_experiment_processor.R` melletti `utils` mappában van. Fájljaink a következőképp néznének ki ez esetben:

`functions.R`:

```
write.csv.custom <- function(x, file, quote=F, sep=',', row.names=F, ...) {
  write.table(x, file, quote=quote, sep=sep, row.names=row.names, ...)
}

write.result <- function(x, path, filename, num='') {
  path <- paste(path, filename, sep='')
  destination <- paste(path, '.csv', sep=as.character(num))
  write.csv.custom(x, destination)
}
```

`chinese_experiment_processor.R`:

```
source('utils/functions.R')

for (i in 1:100){
  file.name <- paste('4/experiments/experiment', '.csv', sep=as.character(i))
  exp <- read.csv(file.name)

  exp$life.span <- exp$died - exp$born

  out.file <- paste('4/results/result', '.csv', sep=as.character(i))
  write.result (exp, 'result')

  summary.file <- paste('4/results/summary', '.csv', sep=as.character(i))
  write.result (summary(exp), 'summary')

  means.list <- list(life.span=mean(exp$life.span), weight=mean(exp$weight))
  write.table(means.list, 'means', quote = F, sep=',', row.names = F)
}
```

Tesztelés

Függvényeket igen sok okból megéri írni:

- Ha egy probléma sokszor fölmerül, sokszor copy-past-elhetjük a megoldására írt kódot. Viszont, ha kiderül, hogy a megoldás helytelen, akkor annyiszor kell átírnunk, ahányszor beillesztettük. Ha függvényt írunk, elég csupán a függvényt átírni, hiszen innentől a hivás összes helyén a megfelelő algoritmust hívjuk.
- A függvények nevet adnak egy adott kód blokknak. Ezáltal nem csupán újrafelhazsnálható kódot kapunk, de gyakorlatilag egy "fejezet nevet" is.
- A visszatérési értékkel rendelkező függvények automatizáltan tesztelhetőek.

Ha van egy bármilyen függvényünk azt azzal a céllal írjuk, hogy adott bementre adott kimenetet kapjunk. Ezt mi fejben is végig tudjuk játszani, így hogy kipróbáljuk, hogy jól működik-e beadunk neki egy tetszőleges értéket, majd megnézzük, hgoy azt vártuk-e, amit kaptunk. Ha nem, átírjuk a függvényt, majd újra elvégezzük a tesztet, amíg helyes eredményt nem kapunk.

Ezt automatizálni is tudjuk **unittesetek** segítéségvel. Azok, akik már régóta kódolnak utánanézhetnek a `testthat` csomagnak. Akik nem azok először gyakorolják a programozást, a unittest ugyanis egy olyan program, amit azért írunk, hogy ellenőrizzen egy általunk írt másik prgramot. Ezt ne próbáljuk meg a élnül, hogy magabiztosan és önállóan tudnánk programozni!

Ettől függetlenül érdemes már gyakorlás során is úgy íni a függvényeinket, hogy azok később tesztelhetőek legyenek. Ehhez az kell hogy:

- A függvényeinknek legyen visszatérési értékük. A `write.result`-ot például át kéne írnunk úgy, hogy az összeállított path stringet visszadja. Ezt már tudjuk tesztelni az alapján, hogy beadunk neki path-t, fájlenvet és kiterjesztést, majd megnézzük, hogy megfelelő string-et kapunk-e vissza.
- Egy függvényünk egyszerre kevés dolgot csinál. Ez a **single responsibility principle**. Ez nem azt jelenti, hogy egy függvényben egy sor utasítás van, hanem azt, hogy minden függvényünk egy egyszerű, egy szószerkezettel könnyen körülhatárolható dolgot végez. Pár ilyen egyszerű függvényt alkalmazhatunk más függvényben, ami ugyan így más több utasítást hajt végre (hiszen a saját maga által meghívott függvények utasításai is lefutnak), de már egy magasabb absztarkciós szinten hajt végre egy jól körülhatárolható dolgot. Az alábbi példában a `simple.abstract.function` még mindig egy egyszerű, jól behatárolható dolgot csinál, de már a két egyszerű matematikai függvény segítségével.

vagyis:

```
simple.mathematical.function <- function(x, y){
  # simple math here
}

other.simple.mathematical.function <- function(x, y){
  # simple math here
}

simple.abstract.function <- function(abstract.object){
  a <- simple.mathematical.function(abstract.object[,1], abstract.object[,2])
  b <- other.simple.mathematical.function(35, abstract.object[1,3])
  return(a + b)
}
```

Egyszerű ökölszabályként mondhatjuk, hogy egy függvény lehetőleg sose tartalmazzon többet 5 sornál, 10 sornál többet pdig csak igen kivételes esetekben! Persze ehhez azért nem kell görcsösen ragaszkodni.