

Bevezetés az R nyelv és statisztikai számítási környezet használatába

Ferenci Tamás

2025. december 9.

Tartalomjegyzék

Előszó	4
1 R szkriptek és az RStudio	6
2 Adattípusok, adatszerkezetek	9
2.1 Értékadás	9
2.2 Adattípusok	12
2.2.1 Numerikus	12
2.2.2 Szöveg	13
2.2.3 Logikai	14
2.2.4 Az adattípusokhoz kapcsolódó néhány fontos művelet	15
2.3 Adatszerkezetek és indexelés	17
2.3.1 Vektor	17
2.3.2 Mátrix	22
2.3.3 Tömb (array)	23
2.3.4 Data frame	24
2.3.5 Lista	34
3 Függvények	38
3.1 Paraméterátadás és visszatérési érték	39
3.2 Saját függvények létrehozása	41
3.3 R csomagok használata	43
3.4 Függvények specifikációja és meghívása	47
3.5 Az R-ben minden utasítás függvényhívás	51
4 Az R programozása	59
4.1 Ismétlődő feladat kezelése saját függvénnyel	59
4.2 Ciklusszervezés	61
4.3 Funkcionális programozás	65
5 Data table: egy továbbfejlesztett adatkeret	67
5.1 Sebesség és nagyméretű adatbázisok kezelése	69
5.2 Jobb kiíratás	70
5.3 Kényelmesebb sorindexelés (sor-szűrés és -rendezés)	71
5.4 Kibővített oszlopindexelés: oszlop-kiválasztás és oszlop-létrehozás műveletekkel	75
5.5 Csoportosítás (aggregáció)	82

5.6	Indexelések láncolása egymás után	91
5.7	Referencia szemantika	95

Előszó

Az R egy ingyenes, nyílt forráskódú, rendkívüli tudású és folyamatosan fejlődő programozási nyelv illetve statisztikai számítási környezet, mely kiválóan alkalmas a legkülönbözőbb statisztikai és adattudományi feladatok megoldására.

Az R egyik fontos jellemzője, hogy lényegében minden feladat elvégzéséhez egy szkriptet kell írunk – szemben más statisztikai programokkal¹, ahol csak egy grafikus felületen kell kattintgatnunk. Ez elsőre ijesztőnek hangozhat, és csakugyan igaz, hogy más programokhoz képest a tanulási görbe meredekebben indul, hiszen a kattintgatással szemben itt már két szám átlagolásához is programot kell írni. A dolog azonban kifizetődő: lehet, hogy egyszerű dolgokat más statisztikai környezetekben könnyebb végrehajtani, itt meg bonyolultabb, de cserében itt a bonyolultabbakat sem sokkal nehezebb, míg más statisztikai programokban az, vagy egyenesen lehetetlen. Kicsit is komolyabb elemzések, kutatások végzésekor az R megtanulásába befektetett munka hamar – és pláne: busásan – megtérül.

A fentiekből már érthető, hogy ahhoz, hogy el tudjunk kezdeni statisztikai elemzéseket végezni R-ben, először az R-rel mint programozási nyelvvel kell megismerkedni. Nagyon fontos hangsúlyozni, hogy ez a jegyzet kizárólag az R *nyelvi* kérdéseivel és programozásával foglalkozik, az R *statisztikai* célokra történő felhasználása egy másik jegyzetem (Ferenci Tamás: Bevezetés a biostatistikába³) témája.

Az R talán legnagyobb erejét a hozzá megírt, megszámlálhatatlan sok⁴ kiegészítő csomag adja, amikkel jószerével minden elképzelhető (és számos nehezen elképzelhető...) statisztikai feladat, adott esetben rendkívül bonyolultak is megoldhatóak, sokszor mindössze egy-egy függvényhívással. Számos kitűnő, jól dokumentált kiegészítő csomag érhető el (melyek maguk is ingyenesek és nyílt forráskódúak); nagyon tipikus, hogy a vadonatúj statisztikai módszereket is R-ben implementálják első közlésükkor. Az R csomagok központi repozitóriuma CRAN⁵ (Comprehensive R Archive Network).

Mindezek alapja az R mögött álló, rendkívül széles és erős nemzetközi közösség. Ingyenes programként bárki számára elérhető, nyílt forráskódú programként pedig jól bővíthető, illetve

¹Megjegyzendő, hogy az R-hez is létezik ilyen grafikus felület, az R Commander². Ennek használata komolyabb elemzési feladatok elvégzéséhez nem szükséges, illetve nem hasznos, kezdők számára azonban jó bevezető eszköz lehet, mivel a jól ismert statisztikai programokhoz teljesen hasonló grafikus felülettel ruházza fel az R-et.

³<https://ferenci-tamas.github.io/biostatistika/>

⁴2025 őszén már több mint 22 ezer! Ez szó szerint napról-napra nő.

⁵<https://cran.r-project.org/>

ez sokaknak a tudományra vonatkozó általános filozófiájával – „open science”, nyílt tudomány – is találkozik (így az enyémmel is). Számos statisztikus fejleszt R alá csomagokat, általában nagyon segítőkészek mind az esetleges hibák javításában, mind az új funkciók megvalósítására vonatkozóan. Több fórum érhető el (pl. a Stackoverflow⁶), ahol a kezdőszintű egyszerű problémáktól a legspeciálisabb nehézségekig mindenben segítséget lehet kérni (és nem ritka, hogy a legnevesebb R fejlesztők válaszolnak!). Nagyon sok csomag jelen van a Github⁷-on is, ami szintén kiváló platform az eszmecserére.

Programozási nyelvek „népszerűségét” nem könnyű objektíven lemérni (egyáltalán, definiálni), de azért van pár ezzel kapcsolatos vizsgálat, illetve folyamatosan frissülő mérőszám⁸. A talán legismertebb TIOBE-index⁹ szerint az R 2025 őszén a 13. legnépszerűbb¹⁰ nyelv, a RedMonk 2025. eleji kimutatása szerint a 12. legnépszerűbb¹¹, az IEEE Top Programming Languages 2025-ös kiadása szerint a 11. legnépszerűbb¹². Vegyük figyelembe, hogy a mezőny általános célú nyelveket is tartalmaz, míg az R elég speciális alkalmazási területű.

Az R különösen erős az eredmények kommunikálásban. Kiegészítő csomagokkal könnyedén lehetséges ún. dinamikus dokumentumok készítése, melyek együtt tartalmazzák a kódokat, és a kapcsolódó leírást.

A reprodukálható kutatás jegyében a cikkekkel együtt közzétett elemzések is nagyon gyakran R-ben íródtak, ezekből szintén sok ötlet meríthető.

⁶<https://stackoverflow.com/>

⁷<https://github.com/>

⁸Külön kérdés, hogy a mostani érában mennyi értelme lesz egyáltalán az ilyen indexeknek, ezt egyre több íráspedzegeti.

⁹<https://www.tiobe.com/tiobe-index/>

¹⁰<https://www.tiobe.com/tiobe-index/r/>

¹¹<https://redmonk.com/sogrady/2025/06/18/language-rankings-1-25/>

¹²<https://spectrum.ieee.org/top-programming-languages-2025>

1 R szkriptek és az RStudio

Egy R-ben írt program, gyakrabban használt nevén szkript, R-beli utasítások sorozata. Lehet egyetlen sor, mely két számot átlagol, vagy több ezer utasításból felépülő komplex elemzés. Az R interpretált nyelv, nem fordított, ami azt jelenti, hogy nem a szkript egészét, egyben fordítja le számítógép által végrehajtható kóddá az R, hanem az utasításokat egyesével hajtja végre, utasításról utasításra.

Az RStudio fejlesztői környezet alapbeállításában a bal oldali rész alján látható a konzol, ahol közvetlenül beküldhetünk utasításokat az R-nek, illetve az – akár közvetlenül, akár a lent vázolt módon szkriptből – beküldött utasítások eredményei láthatóak. A konzol felett találjuk a megnyitott szkriptet, vagy szkripteket. Új szkriptet megnyitni (vagy az elsőöt megnyitni, ha még egy sincs nyitva – ez esetben a konzol az egész bal oldalt elfoglalja) a **Ctrl-Shift-N** billentyűkombinációval, vagy az ikonsor bal szélső ikonjára (fehér lap zöld plusz-jellel) kattintva, és ott az **R Script** pontot választva lehet.

A konzolba írt utasítások azonnal végrehajtnak (amint **Enter**-t ütünk, és ezzel beküldjük az utasítást az R-nek), a szkriptbe írt parancsok pedig a **Ctrl-Enter** billentyűkombinációval futtathatóak. (Valójában ez sem mond ellent annak a szabálynak, hogy a konzolba írt dolgok futtatódnak, mert ha jobban megfigyeljük, akkor láthatjuk, hogy a **Ctrl-Enter** igazából csak átmásolja az utasítást a konzolba, majd beküldi.) Ha a szkriptben nincs kijelölve semmi, akkor a **Ctrl-Enter** azt a sort futtatja, amiben a kurzor áll, ha ki van jelölve valami, akkor a kijelölést. (Függetlenül attól, hogy az milyen, lehet több sor is, de egy sor részlete is). Amint volt róla szó, egy utasítás több sorba is átnyúlhat, ez nem okoz problémát, ilyenkor az R megáll, és várja a további sorokat. Az RStudio ezeket szinte mindig felismeri, és okosan jár el: ilyenkor a **Ctrl-Enter** valójában nem egy sort fog beküldeni, hanem az egész utasítást, fontos azonban, hogy ehhez a legelső sorban kell állnunk. Az egész szkript **Ctrl-Alt-R** kombinációval futtatható le, az egész szkript addig a sorig, amiben a kurzor áll, a **Ctrl-Alt-B** kombinációval, az egész szkript az aktuális sortól a végéig **Ctrl-Alt-E** kombinációval futtatható.

Az egyes utasításokat új sorban kell kezdeni (tehát enter-rel kell elválasztani egymástól). Elvileg egy sorba több utasítás is írható, ekkor az egyes utasításokat pontosvesszővel (;) kell elválasztani, de ezt minden körülmények között kerüljük.

Egy utasítás több sorba is átnyúlhat, ezt az R érzékeli, tehát, ha a sor végén még nem záródott be egy utasítás, akkor a következő sorban folytatja a feldolgozást. Azt, hogy új utasítást vár az R, onnan lehet látni, hogy a konzol elején a **>** jel látható. Ha az utasítás nem ér véget a sorban (ezt az R magától érzékeli, például onnan, hogy egy kinyitott zárójel nem lett bezárva a beküldött sorban), akkor automatikusan azt feltételezi, hogy ez azért van, mert a következő

sorban folytatjuk az utasítást. Ilyenkor a konzol elején a > helyett a + jel látható. Ez jelzi, hogy a beküldött utasítást a következő folytatásának tekinti. Amint látja az R, hogy bezárult az utasítás, végrehajtja, és a konzol átugrik újra a > jelre: várja a következő utasítást. Ez a viselkedés egy gyakori hiba forrása: ha beküldünk egy utasítást, amiből véletlenül hagyjuk a záró zárójelet, akkor az R várni fogja a folytatást. Ha azonban ezt nem vesszük észre, és beküldjük a következő utasítást, akkor nem azt fogja végre hajtani (ahogy várnánk), hanem az előző folytatásának tekinti, és úgy próbálja értelmezni. Az eredmény vagy hiba lesz, vagy az, hogy továbbra is + üzemmódban fogja várni az utasításokat, mi pedig nem kapunk eredményt. Ha ilyen történik, tehát küldjük be az utasításokat, amik teljesen helyesek, és mégsem kapunk eredményt, akkor érdemes megnézni, hogy nem + (folytatás) üzemmódban van-e az R. Ha igen, akkor küldjünk be záró zárójelet, ha ezzel sikerül lezárnunk az utasítást, akkor nyilván hibát kapunk, de legalább visszavehetjük az irányítást.

Az aktuálisan szerkesztett szkript **Ctrl-S** utasítással, vagy az ikonsorban a kék színű, egy darab floppy-lemezes ikonra kattintva menthető. A R-szkriptek alapértelmezett kiterjesztése a .R. Fontos, hogy ezt betartsuk, ugyanis az RStudio funkcionalitása csak akkor fog működni, ha a fájlról tudja, hogy az egy R szkript, és ezt a kiterjesztés alapján azonosítja. A **Ctrl-Alt-S** parancs, vagy a kék színű, több floppy-lemezes ikon az összes megnyitott szkriptet menti. Az RStudio képes megőrizni a nem mentett szkripteket is kilépésnél (a nevük **Untitled** majd utána egy sorszám), de erre a lehetőségre azért ne nagyon építsünk, mert egy összeomlásnál elveszhetnek; a biztos a névvel lementett szkript. Mentett szkriptet megnyitni a **Ctrl-O** billentyűparanccsal, vagy az ikonsorban a mappából kifelé mutató zöld nyilas ikonnal lehet.

Minden kicsit is komolyabb munkánkat érdemes szkriptben megírni, hiszen így lesz az elemzési munkafolyamat reprodukálható. A konzolt tipikusan csak gyors, ismétlődően nem igényelt egyszerű számításokhoz használjuk, aminek az eredményére később nem lesz szükségünk, vagy szkriptírás közben az apróbb bizonytalanságok eldöntéséhez (mi is lesz ennek a parancsnak az eredménye?) használjuk.

A kódunkat érdemes kommentelni, hogy később is világos legyen a működése. A komment olyan része a szkriptnek, melyet az R nem hajt végre, hiszen tudja, hogy nem R utasítás, hanem természetes nyelven írt megjegyzés. Ennek elkülönítésére a kommentjel szolgál, ez az R-ben a #: amennyiben az R egy ilyenhez ér, onnantól átugorja a leírtakat egészen a sor végéig. (Ez tehát ún. egysoros kommentjel.) A # az RStudio-ban a **Ctrl-Shift-C**-vel szűrhető be gyorsan: azon sort kommentezi, mégpedig az elejétől fogva, amelyikben a kurzor áll, illetve ha ki van kommentezve, akkor ezt megszünteti. Többsoros kommentre nincs külön jel R-ben, viszont RStudio-ban a **Ctrl-Shift-C** használható több sort kijelölve is, ekkor mindegyiket kommentezi (vagy eltünteti a kommentjelet, ha ki vannak kommentezve).

Az R kisbetű/nagybetű különbségre érzékeny (case sensitive) nyelv, tehát az a és az A nem ugyanaz, két különböző dolog.

Az RStudio nagyon sok eszközzel segíti a kódolást: színekkel jelöli a különböző tartalmú szintaktikai elemeket, elkezdve egy nevet beírni, **Tab**-bal kiegészíti azt (automatikusan, ha csak egy lehetőség van, egy listát ad, ha több is), rövidebb vagy hosszabb sugót jelenít meg

közvetlenül a beírt kód mellett stb. Segíti a kód indentálását: a `Ctrl-I` kombináció szépen beindentálja a kijelölt részt. (Tipikus a `Ctrl-A` majd `Ctrl-I` kombináció: az előbbi kijelöli az egész szkriptet, így tehát ez mindent indentál.)

Az R kódolási stílus kapcsán csak egyetlen megjegyzés előjáróban: vessző után rakjunk szóközt, de nyitó zárójel után, illetve záró zárójel előtt ne.

2 Adattípusok, adatszerkezetek

Az R programozásának megértéséhez szükséges egyik alapelemünk a változó: változóban tudunk információt tárolni, legyen az egyetlen szám vagy egy egész adatbázis, vagy akár egy regressziós modell. Mit jelent az, hogy információt tárolni? Azt, hogy van a memóriában egy helyünk, amit az R lefoglal nekünk – a változó lényegében ezt a memóriahelyet azonosítja – és oda elmenthetünk információt, módosíthatjuk az ott lévő információt, és kiolvashatjuk az ott lévő információt. Változóból tetszőleges számút létrehozhatunk. Elsőként meg kell ismerkednünk a változó fogalmával, a neki történő értékadással, és azzal, hogy milyen típusú adatokat tudunk változóban tárolni.

2.1. Értékadás

Változó értéket az értékadás művelettel kap; ez kb. a „legyen egyenlő” módon olvasható ki. Az értékadás jele az R-ben a `<-`. (A más programnyelveken megszokottabb `=`-t ne használjuk értékadásra, mert bár működne, de az R-es hagyományok szerint ezt egy másik helyzetre tartjuk fent, ezt majd később látni is fogjuk.) A nyíl bal oldalára kerül a változó, a jobb oldalára az érték, amit adni akarunk neki, például (a második sorban a változónév megadása egyszerűen kiírja az értékét, csak hogy lássuk mi történt; hogy miért jelent kiíratást a változónév megadása, arra később még visszatérünk):

```
testtomeg <- 72
testtomeg
```

```
[1] 72
```

Már létező változónak történő értékadásnál az előző érték elveszlik, és felülíródik az aktuálisan megadottal:

```
testtomeg <- 62
testtomeg
```

```
[1] 62
```

Furcsa lehet ez a nyíl jel az értékadás jelölésére, de „didaktikailag” nagyon jó: szépen mutatja, szinte vizuálisan is, hogy a 72 „bemegy” a `testtomeg` változóba. (Elvileg használható a `->` is értékadásra, ilyenkor értelemszerűen fordul a helyzet, de ezt ritkán szokták alkalmazni.)

Ami szembeötlő: ez az utasítás gond nélkül lefut, miközben sehol nem deklaráltuk, hogy a `testtomeg` legyen egy változó, pláne nem adtuk meg, hogy milyen típusú adatot akarunk benne tárolni! Furcsa lehet józan ésszel is, hiszen logikus lenne, hogy előbb valamilyen módon „létre kell hoznunk” egy változót, mielőtt egyáltalán bármit el tudunk tárolni benne. Hogyhogy nem kellett előbb megadni, hogy szeretnénk egy ilyen változót, például azért, hogy az R le tudja foglalni a helyét a memóriában? Azt is várhatnánk, hogy meg kell adni a típusát, tehát, hogy milyen jellegű adatot akarunk benne tárolni (számot, szöveget, dátumot, kis számot, nagy számot stb.), hiszen logikus lenne a feltételezés, hogy ettől függ, hogy mit kell az R-nek tennie, például mennyi memóriát kell foglalnia, tehát okkal várhatnánk, hogy ezt is meg kell adni. De nem! Más programnyelvekben valóban lehet ez a helyzet, de itt nem.

Az R „intelligensen” kitalált mindent: mivel látja, hogy korábban `testtomeg` nevű változó még nem létezett, ezért egyetlen szó nélkül, automatikusan létrehozza, illetve abból, hogy mit adtunk neki értékül, azt is kitalálta, hogy milyen legyen a típusa, jelen esetben szám. Majd természetesen az értékét is beállítja arra, amit megadtunk; ez a szigorúan vett értékadás.

Az „intelligensen” szót nem véletlenül tettem idézőjelbe. Ez egy példa az R egy meglehetősen általános filozófiájára, amire később még sok további példát fogunk látni: az R „megengedi trehánytságot” és igyekszik kitalálni, hogy mit akarhattunk. Bár ez első ránézésre rendkívül kényelmesnek hangzik, fontos hangsúlyozni, hogy ez egy kétélű fegyver! Egyfelől ugyanis valóban nagyon kényelmes, jelen esetben, hogy nem kell törődnünk a változók előzetes deklarálásával, típusuk megadásával, de másrészt így kiesik egy védővonal, ami megóvhatna minket a saját hibáinktól – hiszen a deklaráció rákényszerít(ett volna) minket arra, hogy jobban végiggondoljuk a változókkal kapcsolatos kérdéseket. Így viszont könnyebben előfordulhat, hogy olyat csinálunk, amit igazából nem szeretnénk, ráadásul úgy, hogy észre sem vesszük!

Például, azt írjuk később a kódban, hogy:

```
testtomeg <- 82
```

Mi fog történni? Semmi! Pontosabban szólva, valami nagyon is fog történni, létrejön egy *másik* változó, miközben maga az utasítás egyetlen hang, hiba, figyelmeztetés, minden nélkül lefut! Ha kellett volna előzetesen deklarálni a változókat, akkor ilyen nem fordulhatna elő, hiszen a gép azonnal szólna, hogy `testtomeg` nevű változót nem deklaráltunk.

Vagy egy másik példa. Később azt írjuk (a „később” jelentheti azt, hogy három nappal, és ezer sorral lejjebb, és persze a változó neve sem biztos, hogy ennyire egyértelmű segítséget jelent!), hogy:

```
testtomeg <- "András"
```

Mi történik? Megint csak: semmi, az utasítás egyetlen hang, hiba, figyelmeztetés, minden nélkül lefut, és szó nélkül átállítja a változó típusát, hogy be tudja állítani "András" értékre. Ha kellett volna deklarálni a változó típusát is, az kényelmetlenebb lett volna, igen, de segítene elkerülni az ilyen hibákat, hiszen a gép azonnal szólna, hogy a `testtomeg` változó típusa szám, úgyhogy abba nem rakhatunk bele egy szöveget.

Amint mondtam, a „megengedi a trehányyságot”, ez a kétélű fegyver (kényelmesebb, gyorsabb, de veszélyesebb, kevesebb a hiba elleni védelmi vonal) meglehetősen általános filozófia az R-ben, de ennek a konkrét viselkedésnek külön neve van a programozás-elméletben. Azt mondják, hogy egy programnyelv statikus típusrendszerű, ha a változókat előzetesen deklarálni kell, megadva a típusukat. Ez plusz feladat, de cserében a gép meg tud minket védeni a fentihez hasonló hibáktól. Az R azonban dinamikus típusrendszerű, vagyis a típust nem kell előre megadni¹, és akár, mint láttuk, menet közben is változtatható. Ez sok szempontból egyszerűbb és flexibilisebb, de kinyitja az ajtót a fentihez hasonló hibákhoz.

R-ben a változónév karakterekből, számokból, a `.` és a `_` jelekből állhat, de nem kezdődhet számmal vagy `_` jellel, és ha `.` jellel kezdődik, akkor utána nem jöhet szám. (Bizonyos, úgynevezett foglalt szavakat, amiket az R nyelv használ, nem választhatunk változónévként. Ezekből nagyon kevés van, így annyiban óvatosnak kell lenni, hogy az R egy sor szokásos függvényét simán felüldefiniálhatjunk, ha létrehozunk olyan nevű változót.) Érdekes módon az, hogy az R mit ért karakter alatt, függhet az adott számítógép beállításaitól, de a legbiztosabb, ha a standard latin betűs (ASCII) karaktereket használjuk csak. (Azaz: lehetőleg ne használjunk ékezetes betűt változónévként. Elvileg el lehet vele boldogulni – adott esetben speciális szimbólummal jelölve, hogy az egy változónév – de nem éri meg a veszélyt, csomagokban kiszámíthatatlan gondokat okozhat.)

Egy fontos általános szabály, hogy ha egy utasításban értékadás van, akkor az eltárolás a „háttérben” történik meg, a konzolra nem íródik ki semmi. (Természetesen vannak kivételek, olyan számítások, amik mellékhatásként mindenképp kiírnak valamit a konzolra.) Értékadás nélküli utasítás futtatásánál viszont épp fordított a helyzet: az eredmény kiíratódik a konzolra, de nem tárolódik el sehol. Ha egy értékadást gömbölyű zárójelekbe ágyazunk (`(a <- 1)`), akkor el is tárolódik és ki is íratódik az eredmény; a gyakorlatban ritkán használjuk.

(Egy apró jótanács. Mi van akkor, ha lefuttatunk egy rendkívül hosszú utasítást, de véletlenül elfelejtjük benyilazni egy változóba... azaz az eredmény megjelenik a konzolon, viszont nem tárolódott le! Most futtathatjuk az egészet újra?! Szerencsére nem: az R valójában nyíl nélkül

¹Ez nem feltétlenül jelenti azt, hogy a programnyelv semmilyen ellenőrzést nem végez a tekintetben, hogy a típusokat helyesen kezeljük-e. Szoktak beszélni erősen és gyengén típusos nyelvekről a szerint, hogy mennyire szigorú ellenőrzéseket végez a programnyelv, de ezeket a fogalmakat nagyon sokféleképp definiálták, és nem egyértelmű, hogy mit jelentenek. Ha valaki azt mondja, hogy az R gyengén típusos, akkor valószínűleg arra gondol, hogy nagyon megengedőn és könnyedén konvertál különböző típusokat, implicite is, tehát ha nem mi írjuk konkrétan elő a konverziót – ezt később részletesen is fogjuk látni.

is eltárolja egy speciális változóban az eredményt, a neve `.Last.value`. Ha tehát ilyen történik, akkor ne essünk kétségbe, ezt speciális változót adjuk értékül a változónknak. De vigyázzunk, ilyen módon mindig csak a legutóbbi utasítás eredménye érhető el.)

2.2. Adattípusok

Elsőként meg kell ismerkednünk azzal, hogy a korábban említett típusok pontosan milyenek lehetnek – ez lényegében azt adja meg, hogy milyen jellegű adatot tárolunk az adott változóban. Az R-ben 4 fontos adattípus van: numerikus, amelybe a valós és az egész típusok tartoznak alcsoportként, a szöveg és a logikai. (Elvileg még két további típus létezik, a `complex` és a `raw`, ezek nagyon ritkán használatosak.) Létezik még egy fogalom, a `factor`, ami adattípusnak tűnik, de mégsem az (egy másik típus speciális esete), erről később fogunk szót ejteni.

A változó típusát az R többféle módon is értelmezi, de a gyakorlatban inkább az `str` függvény ismerete a fontosabb, mellyel komplexebb adatszerkezetekről is jól áttekinthető információt tudunk nyerni.

2.2.1. Numerikus

Számok tárolására a numerikus típus (`numeric`, rövidítve `num`) szolgál.

Alapbeállításban ez a típus valós számokat tárol (precízen: double pontosságú lebegőpontos). A double pontossága jellemzően 53 bit (kb. $2 \cdot 10^{-308}$ -tól $2 \cdot 10^{308}$ -ig nagyjából $2 \cdot 10^{-16}$ felbontással; az adott architektúra vonatkozó értéket a `.Machine` megmondja).

Az R-ben a tizedestörtöket angol stílusban kell megadni, tehát a tizedesjelölő a pont, nem a vessző.

Így néz ki egy numerikus adattal történő értékadás:

```
szam <- 3.1
szam
```

```
[1] 3.1
```

```
str(szam)
```

```
num 3.1
```

Nézzük meg, hogy csakugyan case sensitive a nyelv:

```
SZAM
```

```
Error: object 'SZAM' not found
```

```
Szam
```

```
Error: object 'Szam' not found
```

```
szaM
```

```
Error: object 'szaM' not found
```

Fontos megjegyezni, hogy attól mert valami történetesen egész, az R még nem fogja egész számként kezelni, ugyanúgy valósnak veszi:

```
szam <- 3  
str(szam)
```

```
num 3
```

Ha egészet (integer) akarunk, azt explicite jelölni kell a szám után fűzött L utótaggal:

```
egesz <- 3L  
egesz
```

```
[1] 3
```

```
str(egesz)
```

```
int 3
```

2.2.2. Szöveg

Szemben más programnyelvekkel, az R-ben nincs megkülönböztetve az egy karakter, és a több karakterből álló karakterfüzér (sztring). Számára mindkettő ugyanolyan típusú (**character**, rövidítve **chr**).

A szöveget idézőjelek közé kell tenni, ebből tudja az R, hogy az egy – szöveget tartalmazó – konstans, és nem egy kiértékelendő kifejezés (különben a **kiskutya** beírásakor egy ilyen nevű változót kezdene keresni az R):

```
szoveg <- "kiskutya"  
szoveg
```

```
[1] "kiskutya"
```

```
str(szoveg)
```

```
chr "kiskutya"
```

```
typeof(szoveg)
```

```
[1] "character"
```

A sztringkonstansokat idézőjellel kell jelölni. Az R megengedi a dupla (" ") és a szimpla (' ') idézőjel használatát is, de az előbbi a preferált (az R általi kiírás is mindenképp ilyenl történik), az utóbbit érdemes az egymásbaágyazott esetekre használni (tehát, ha egy sztringkonstans tartalmaz egy idézőjeles részt²). Természetesen az "1" kifejezés nem az 1 számot, hanem az 1-et (mint karaktert) tartalmazó sztringet jelenti.

Az RStudio-ban a szintaxis highlighting segít, a szövegek alapértelmezés szerint zöld színnel jelennek meg.

A szövegben elhelyezhetünk különféle speciális jeleket is, mint a tabulátor vagy sortörés, ezeket backslash jelöli (például a tabulátor `\t`).

<https://www.youtube.com/watch?v=QiTaaQFhPJc>

2.2.3. Logikai

Logikai (`logical`, rövidítve `logi`) típusú változóban bináris, azaz igaz/hamis (igen/nem) értékeket tárolhatunk. A két értéket foglalt szavak jelzik, az igazat a `TRUE`, a hamisat a `FALSE` (a case sensitivity miatt természetesen fontos, hogy csupa nagybetű!):

```
logikai <- TRUE  
logikai
```

```
[1] TRUE
```

²Ellenkező esetben a escape-elni kellene a dupla idézőjelet – egy elé írt backslash-sel, `\"` formában – különben az R nem tudhatná, hogy az nem a szöveg végét jelenti.

```
str(logikai)
```

```
logi TRUE
```

```
typeof(logikai)
```

```
[1] "logical"
```

A TRUE rövidíthető T-nek, a FALSE pedig F-nek.

Természetesen ilyen bináris adatokat nyugodtan tárolhatnánk numerikus változóként is (például 0 és 1 formájában), de a logikai változó előnye, hogy van szemantikája, azaz maga az adattípus is kifejezi, hogy az egyes értékek mit jelentenek, igazat és hamisat (nem pedig számokat), ez sokszor kényelmesebb és tisztább. Emellett értelemszerűen a memóriaigénye is kisebb, bár ennek a legtöbb esetben valószínűleg nincs érdemi jelentősége.

<https://www.youtube.com/watch?v=FuBE1Csgmi4>

2.2.4. Az adattípusokhoz kapcsolódó néhány fontos művelet

Adott típus tesztelése az `is.<típus>` alakban lehet:

```
is.integer(szam)
```

```
[1] FALSE
```

```
is.integer(egesz)
```

```
[1] TRUE
```

```
is.integer(szoveg)
```

```
[1] FALSE
```

```
is.integer(logikai)
```

```
[1] FALSE
```

Az `is.numeric` azt jelenti, hogy `is.integer` vagy `is.double`:

```
is.double(szam)
```

```
[1] TRUE
```

```
is.double(egesz)
```

```
[1] FALSE
```

```
is.numeric(szam)
```

```
[1] TRUE
```

```
is.numeric(egesz)
```

```
[1] TRUE
```

Adott típussá alakítás `as.<tipus>` alakban lehet:

```
as.character(szam)
```

```
[1] "3"
```

```
as.numeric(szoveg)
```

Warning: NAs introduced by coercion

```
[1] NA
```

```
as.numeric("2.4" )
```

```
[1] 2.4
```

```
as.numeric(logikai)
```

```
[1] 1
```


A sémát már a fentiek is mutatják: a konvertálásnál egy „erőssorrend”, jelesül `character < double = integer < logical`, amely irányban mindig lehet konvertálni (a T 1-re, a F 0-ra alakul, a többi értelemszerű). A sorrenddel ellentétesen is elképzelhető, hogy lehet konvertálni, de ez már nem biztos, azon múlik, hogy értelmesen végrehajtható-e (a "kiskutya" nem konvertálható számmá, az "1" igen). Sok függvény automatikusan konvertál, például ha egy logikai igaz értékhez hozzáadunk 1-et, akkor 2-t kapunk, mert a háttérben, szó nélkül, át fogja konvertálni számmá.

A sikertelen konverziók NA-t adnak, amely az R-ben lényegében a „hiányzó érték” jele.

Speciális szerepe van még a NULL-nak (ez inkább olyasmit jelöl, hogy „üres objektum”), illetve az NaN-nek (not-a-number, tipikusan olyan adja, mint például ha negatív szám logaritmusát vesszük).

2.3. Adatszerkezetek és indexelés

Most, hogy ismerjük az adattípusokat, azzal kell folytatnunk, hogy ezekből milyen komplexebb struktúrák rakhatóak össze.

2.3.1. Vektor

A vektor homogén, egydimenziós adatszerkezet. Egydimenziós, mert egy „kiterjedése” van, egy indexszel hivatkozhatunk az elemeire, és homogén, mert minden benne lévő adat ugyanolyan típusú kell legyen. Szemben a „vektor” matematikai fogalmával, nem kötelező, hogy ezek számok legyenek, de mindenképp ugyanolyannak kell lennie a típusuknak.

Vektor legegyszerűbb módon az elemei felsorolásával hozható létre, ehhez a `c` függvény használható:

```
szamvektor <- c(1, 4, 5, -2, 3.5, 10)
szamvektor
```

```
[1] 1.0 4.0 5.0 -2.0 3.5 10.0
```

Sok függvény vektort ad vissza eredményül, például a `seq`-val generálhatunk egy reguláris sorozatot. A függvényekről később lesz szó, úgyhogy most kommentár nélkül: a `seq(1, 101, 2)` hívás kidobja a számokat 1-től 101-ig 2-esével:

```
seq(1, 101, 2)
```

```
[1] 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37
[20] 39 41 43 45 47 49 51 53 55 57 59 61 63 65 67 69 71 73 75
[39] 77 79 81 83 85 87 89 91 93 95 97 99 101
```

Az eredmény egy vektor.

Arra a speciális esetre, hogy 1-esével lépkedünk, olyan sűrűn van szükség, hogy arra van egy külön, rövidebb jelölés, a `::`:

```
1:100
```

```
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
[19] 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
[37] 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54
[55] 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
[73] 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90
[91] 91 92 93 94 95 96 97 98 99 100
```

A sorok elején lévő, szögletes zárójelbe írt számok nem részei a vektornak, az az olvashatóságot segíti: ha nagyon hosszú vektorban kell egy adott elem pozícióját megtalálni, akkor nem a legelejétől kell számolni, elég a sor elejétől menni.

Feltűnhet, hogy a korábbi `szam` kiíratás esetén is megjelent egy `[1]` a sor elején. Ez nem véletlen: a valóságban „skalár” nincs az R-ben, igazából a `szam` is egy vektor (csak épp egy elemből áll).

Ahogy volt róla szó, nem csak numerikus adatokból képezhető vektor, hanem bármilyenből:

```
karaktervektor <- c("a", "b", "xyz")
karaktervektor
```

```
[1] "a" "b" "xyz"
```

A vektor homogén, ezért az alábbi utasítások csak és kizárólag azért futnak le mégis, mert a háttérben ilyenkor az R a „leggyengébbre” konvertálja az összeset (hogy kikényszerítse a homogenitást):

```
c(1, "a")
```

```
[1] "1" "a"
```

```
c(2, TRUE)
```

```
[1] 2 1
```

A vektor elemei el is nevezhetőek; a nevek később a `names`-zel lekérhetőek:

```
szamvektor <- c(elso = 4, masodik = 1, harmadik = 7)
szamvektor
```

```
    elso  masodik harmadik
      4         1         7
```

```
names(szamvektor)
```

```
[1] "elso"      "masodik"    "harmadik"
```

A `names` érdekesen viselkedik, mert nem csak megadja a neveket, de bele is nyilazhatunk értéket, ez esetben beállítja:

```
names(szamvektor) <- c("egy", "ketto", "harom")
szamvektor
```

```
    egy ketto  harom
      4      1      7
```

Az adatszerkezetek esetén egy alapvető kérdés az indexelés, tehát, hogy hogyan hivatkozhatunk adott pozícióban lévő elemre vagy elemekre. Ennek az R-ben meglehetősen sok módja lehetséges, de általános, hogy az indexelést a szögletes zárójel jelöli. (Később fogunk még egy szintaktikai elemet látni indexelésre.)

A legegyszerűbb eset, ha egyetlen számmal indexelünk: ekkor az adott pozícióban lévő elemet kapjuk meg. Például:

```
szamvektor[3]
```

```
harom
      7
```

Megtehetjük azt is, hogy nem egy számot, hanem egy vektort adunk át, ekkor a felsorolt pozícióban lévő elemeket kapjuk, a felsorolás sorrendjében:

```
szamvektor[c(1, 3)]
```

```
egy három  
4      7
```

(Ugye látjuk, hogy ez a kettő igazából ugyanaz? Az előbbi példa is vektorral indexeltm hiszen „egy szám” nincsen, az is vektor.)

Egy elem kiválasztható többször is, illetve tetszőleges sorrendben:

```
szamvektor[c(2, 2, 1, 3, 2, 3, 1, 1)]
```

```
ketto ketto    egy három ketto három    egy    egy  
1      1      4      7      1      7      4      4
```

Nemlétező elem indexelése NA-t ad:

```
szamvektor[10]
```

```
<NA>  
NA
```

A második alapvető megoldás a logikai vektorral való indexelés: ekkor egy ugyanolyan hosszú vektort kell átadnunk, mint az indexelendő vektor, és azokat az elemeket választja ki, ahol logikai igaz érték van:

```
szamvektor[c(TRUE, FALSE, TRUE, TRUE, FALSE, TRUE)]
```

```
egy három <NA> <NA>  
4      7    NA    NA
```

Valójában azonban ez is működik, hiába rövidebb az indexelő vektor:

```
szamvektor[c(TRUE, TRUE, FALSE)]
```

```
egy ketto  
4      1
```

Ez egy újabb példa a kétélű flexibilitásra: azért fog működni, mert ilyenkor az R „reciklálja” az indexelő vektort.

Lehetséges negatív indexelés is, ez kiválaszt mindent, *kivéve* amit indexeltünk:

```
szamvektor[-3]
```

```
egy ketto  
  4      1
```

```
szamvektor[-c(1, 3)]
```

```
ketto  
  1
```

Ha vannak elnevezések, akkor azok használhatóak indexelésre is:

```
szamvektor["masodik"]
```

```
<NA>  
NA
```

```
szamvektor[c("masodik", "utolso")]
```

```
<NA> <NA>  
NA   NA
```

Az indexelés és az értékadás kombinálható is:

```
szamvektor[ 3 ] <- 99  
szamvektor
```

```
egy ketto harm  
  4      1    99
```

```
szamvektor[ 10 ]
```

```
<NA>  
NA
```

Ha nemlétezőnek adunk értéket, automatikusan kiterjeszti a vektort, a többi helyre pedig NA kerül (megint újabb példa a kétélű flexibilitásra):

```
szamvektor[ 10 ] <- 999
szamvektor
```

```
    egy ketto  három
4      1      99    NA    NA    NA    NA    NA    999
```

2.3.2. Mátrix

A mátrix homogén, kétdimenziós adatszerkezet.

Legegyszerűbben úgy tölthető fel, ha egy vektort áttördelünk, a `matrix` függvény használatával (az `nc` argumentummal az oszlopok, az `nr` argumentummal a sorok számát állíthatjuk be, értelemszerűen elég a kettőből egyet megadni):

```
szammatrix <- matrix( 1:6, nc = 2 )
szammatrix
```

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

Alapból oszlopok szerint tördel, de a `byrow` argumentummal ezt átállíthatjuk:

```
matrix( 1:6, nc = 2, byrow = TRUE )
```

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

A dimenzió, illetve külön a sorok és oszlopok száma könnyen lekérhető:

```
dim( szammatrix )
```

```
[1] 3 2
```

```
nrow( szammatrix )
```

```
[1] 3
```

```
ncol( szammatrix )
```

```
[1] 2
```

A mátrix oszlopai és sorai is elnevezhetőek, emiatt itt nem egy **names** van, hanem egy **row.names** és egy **names**, ez utóbbi az oszlopnév, de egyebekben teljesen hasonlóan viselkednek.

Indexelés ugyanúgy végezhető, csak épp mindkét dimenzióra mondanunk kell valamit; a kettő vesszővel választandó el:

```
szammatrix[ c( 2, 3 ), 2 ]
```

```
[1] 5 6
```

Mindkét dimenzió tetszőleges korábban látott módon indexelhető, tehát a különböző módok keverhetőek is:

```
szammatrix[ c( 1, 2 ), c( T, F ) ]
```

```
[1] 1 2
```

Ha egy dimenziót nem indexelünk, akkor az R úgy érti, hogy onnan minden elem (de a vessző ekkor sem hagyható el!):

```
szammatrix[ 2, ]
```

```
[1] 2 5
```

2.3.3. Tömb (array)

A tömb (array) homogén, n -dimenziós adatszerkezet (nem foglalkozunk vele részletesebben, ritkán használatos).

2.3.4. Data frame

A data frame (adatkeret) heterogén, kétdimenziós, rektanguláris adatszerkezet. Pontosabban szólva félig heterogén: az oszlopok homogének, de a különböző oszlopok típusai eltérhetnek egymástól. Lényegében tehát - nem feltétlenül ugyanolyan típusú - vektorok összefogva; a rektanguláris azt jelenti, hogy minden vektor ugyanolyan hosszú kell legyen.

Ez a legtipikusabb adatszerkezet orvosi adatok tárolására: sorokban a megfigyelési egységek, oszlopokban a változók.

A `data` paranccsal egy kiegészítő csomagban található kész adat tölthető be:

```
data( birthwt, package = "MASS" )  
birthwt
```

	low	age	lwt	race	smoke	ptl	ht	ui	ftv	bwt
85	0	19	182	2	0	0	0	1	0	2523
86	0	33	155	3	0	0	0	0	3	2551
87	0	20	105	1	1	0	0	0	1	2557
88	0	21	108	1	1	0	0	1	2	2594
89	0	18	107	1	1	0	0	1	0	2600
91	0	21	124	3	0	0	0	0	0	2622
92	0	22	118	1	0	0	0	0	1	2637
93	0	17	103	3	0	0	0	0	1	2637
94	0	29	123	1	1	0	0	0	1	2663
95	0	26	113	1	1	0	0	0	0	2665
96	0	19	95	3	0	0	0	0	0	2722
97	0	19	150	3	0	0	0	0	1	2733
98	0	22	95	3	0	0	1	0	0	2751
99	0	30	107	3	0	1	0	1	2	2750
100	0	18	100	1	1	0	0	0	0	2769
101	0	18	100	1	1	0	0	0	0	2769
102	0	15	98	2	0	0	0	0	0	2778
103	0	25	118	1	1	0	0	0	3	2782
104	0	20	120	3	0	0	0	1	0	2807
105	0	28	120	1	1	0	0	0	1	2821
106	0	32	121	3	0	0	0	0	2	2835
107	0	31	100	1	0	0	0	1	3	2835
108	0	36	202	1	0	0	0	0	1	2836
109	0	28	120	3	0	0	0	0	0	2863
111	0	25	120	3	0	0	0	1	2	2877
112	0	28	167	1	0	0	0	0	0	2877
113	0	17	122	1	1	0	0	0	0	2906

114	0	29	150	1	0	0	0	0	2	2920
115	0	26	168	2	1	0	0	0	0	2920
116	0	17	113	2	0	0	0	0	1	2920
117	0	17	113	2	0	0	0	0	1	2920
118	0	24	90	1	1	1	0	0	1	2948
119	0	35	121	2	1	1	0	0	1	2948
120	0	25	155	1	0	0	0	0	1	2977
121	0	25	125	2	0	0	0	0	0	2977
123	0	29	140	1	1	0	0	0	2	2977
124	0	19	138	1	1	0	0	0	2	2977
125	0	27	124	1	1	0	0	0	0	2922
126	0	31	215	1	1	0	0	0	2	3005
127	0	33	109	1	1	0	0	0	1	3033
128	0	21	185	2	1	0	0	0	2	3042
129	0	19	189	1	0	0	0	0	2	3062
130	0	23	130	2	0	0	0	0	1	3062
131	0	21	160	1	0	0	0	0	0	3062
132	0	18	90	1	1	0	0	1	0	3062
133	0	18	90	1	1	0	0	1	0	3062
134	0	32	132	1	0	0	0	0	4	3080
135	0	19	132	3	0	0	0	0	0	3090
136	0	24	115	1	0	0	0	0	2	3090
137	0	22	85	3	1	0	0	0	0	3090
138	0	22	120	1	0	0	1	0	1	3100
139	0	23	128	3	0	0	0	0	0	3104
140	0	22	130	1	1	0	0	0	0	3132
141	0	30	95	1	1	0	0	0	2	3147
142	0	19	115	3	0	0	0	0	0	3175
143	0	16	110	3	0	0	0	0	0	3175
144	0	21	110	3	1	0	0	1	0	3203
145	0	30	153	3	0	0	0	0	0	3203
146	0	20	103	3	0	0	0	0	0	3203
147	0	17	119	3	0	0	0	0	0	3225
148	0	17	119	3	0	0	0	0	0	3225
149	0	23	119	3	0	0	0	0	2	3232
150	0	24	110	3	0	0	0	0	0	3232
151	0	28	140	1	0	0	0	0	0	3234
154	0	26	133	3	1	2	0	0	0	3260
155	0	20	169	3	0	1	0	1	1	3274
156	0	24	115	3	0	0	0	0	2	3274
159	0	28	250	3	1	0	0	0	6	3303
160	0	20	141	1	0	2	0	1	1	3317
161	0	22	158	2	0	1	0	0	2	3317

162	0	22	112	1	1	2	0	0	0	3317
163	0	31	150	3	1	0	0	0	2	3321
164	0	23	115	3	1	0	0	0	1	3331
166	0	16	112	2	0	0	0	0	0	3374
167	0	16	135	1	1	0	0	0	0	3374
168	0	18	229	2	0	0	0	0	0	3402
169	0	25	140	1	0	0	0	0	1	3416
170	0	32	134	1	1	1	0	0	4	3430
172	0	20	121	2	1	0	0	0	0	3444
173	0	23	190	1	0	0	0	0	0	3459
174	0	22	131	1	0	0	0	0	1	3460
175	0	32	170	1	0	0	0	0	0	3473
176	0	30	110	3	0	0	0	0	0	3544
177	0	20	127	3	0	0	0	0	0	3487
179	0	23	123	3	0	0	0	0	0	3544
180	0	17	120	3	1	0	0	0	0	3572
181	0	19	105	3	0	0	0	0	0	3572
182	0	23	130	1	0	0	0	0	0	3586
183	0	36	175	1	0	0	0	0	0	3600
184	0	22	125	1	0	0	0	0	1	3614
185	0	24	133	1	0	0	0	0	0	3614
186	0	21	134	3	0	0	0	0	2	3629
187	0	19	235	1	1	0	1	0	0	3629
188	0	25	95	1	1	3	0	1	0	3637
189	0	16	135	1	1	0	0	0	0	3643
190	0	29	135	1	0	0	0	0	1	3651
191	0	29	154	1	0	0	0	0	1	3651
192	0	19	147	1	1	0	0	0	0	3651
193	0	19	147	1	1	0	0	0	0	3651
195	0	30	137	1	0	0	0	0	1	3699
196	0	24	110	1	0	0	0	0	1	3728
197	0	19	184	1	1	0	1	0	0	3756
199	0	24	110	3	0	1	0	0	0	3770
200	0	23	110	1	0	0	0	0	1	3770
201	0	20	120	3	0	0	0	0	0	3770
202	0	25	241	2	0	0	1	0	0	3790
203	0	30	112	1	0	0	0	0	1	3799
204	0	22	169	1	0	0	0	0	0	3827
205	0	18	120	1	1	0	0	0	2	3856
206	0	16	170	2	0	0	0	0	4	3860
207	0	32	186	1	0	0	0	0	2	3860
208	0	18	120	3	0	0	0	0	1	3884
209	0	29	130	1	1	0	0	0	2	3884

210	0	33	117	1	0	0	0	1	1	3912
211	0	20	170	1	1	0	0	0	0	3940
212	0	28	134	3	0	0	0	0	1	3941
213	0	14	135	1	0	0	0	0	0	3941
214	0	28	130	3	0	0	0	0	0	3969
215	0	25	120	1	0	0	0	0	2	3983
216	0	16	95	3	0	0	0	0	1	3997
217	0	20	158	1	0	0	0	0	1	3997
218	0	26	160	3	0	0	0	0	0	4054
219	0	21	115	1	0	0	0	0	1	4054
220	0	22	129	1	0	0	0	0	0	4111
221	0	25	130	1	0	0	0	0	2	4153
222	0	31	120	1	0	0	0	0	2	4167
223	0	35	170	1	0	1	0	0	1	4174
224	0	19	120	1	1	0	0	0	0	4238
225	0	24	116	1	0	0	0	0	1	4593
226	0	45	123	1	0	0	0	0	1	4990
4	1	28	120	3	1	1	0	1	0	709
10	1	29	130	1	0	0	0	1	2	1021
11	1	34	187	2	1	0	1	0	0	1135
13	1	25	105	3	0	1	1	0	0	1330
15	1	25	85	3	0	0	0	1	0	1474
16	1	27	150	3	0	0	0	0	0	1588
17	1	23	97	3	0	0	0	1	1	1588
18	1	24	128	2	0	1	0	0	1	1701
19	1	24	132	3	0	0	1	0	0	1729
20	1	21	165	1	1	0	1	0	1	1790
22	1	32	105	1	1	0	0	0	0	1818
23	1	19	91	1	1	2	0	1	0	1885
24	1	25	115	3	0	0	0	0	0	1893
25	1	16	130	3	0	0	0	0	1	1899
26	1	25	92	1	1	0	0	0	0	1928
27	1	20	150	1	1	0	0	0	2	1928
28	1	21	200	2	0	0	0	1	2	1928
29	1	24	155	1	1	1	0	0	0	1936
30	1	21	103	3	0	0	0	0	0	1970
31	1	20	125	3	0	0	0	1	0	2055
32	1	25	89	3	0	2	0	0	1	2055
33	1	19	102	1	0	0	0	0	2	2082
34	1	19	112	1	1	0	0	1	0	2084
35	1	26	117	1	1	1	0	0	0	2084
36	1	24	138	1	0	0	0	0	0	2100
37	1	17	130	3	1	1	0	1	0	2125

40	1	20	120	2	1	0	0	0	3	2126
42	1	22	130	1	1	1	0	1	1	2187
43	1	27	130	2	0	0	0	1	0	2187
44	1	20	80	3	1	0	0	1	0	2211
45	1	17	110	1	1	0	0	0	0	2225
46	1	25	105	3	0	1	0	0	1	2240
47	1	20	109	3	0	0	0	0	0	2240
49	1	18	148	3	0	0	0	0	0	2282
50	1	18	110	2	1	1	0	0	0	2296
51	1	20	121	1	1	1	0	1	0	2296
52	1	21	100	3	0	1	0	0	4	2301
54	1	26	96	3	0	0	0	0	0	2325
56	1	31	102	1	1	1	0	0	1	2353
57	1	15	110	1	0	0	0	0	0	2353
59	1	23	187	2	1	0	0	0	1	2367
60	1	20	122	2	1	0	0	0	0	2381
61	1	24	105	2	1	0	0	0	0	2381
62	1	15	115	3	0	0	0	1	0	2381
63	1	23	120	3	0	0	0	0	0	2410
65	1	30	142	1	1	1	0	0	0	2410
67	1	22	130	1	1	0	0	0	1	2410
68	1	17	120	1	1	0	0	0	3	2414
69	1	23	110	1	1	1	0	0	0	2424
71	1	17	120	2	0	0	0	0	2	2438
75	1	26	154	3	0	1	1	0	1	2442
76	1	20	105	3	0	0	0	0	3	2450
77	1	26	190	1	1	0	0	0	0	2466
78	1	14	101	3	1	1	0	0	0	2466
79	1	28	95	1	1	0	0	0	2	2466
81	1	14	100	3	0	0	0	0	2	2495
82	1	23	94	3	1	0	0	0	0	2495
83	1	17	142	2	0	0	1	0	0	2495
84	1	21	130	1	1	0	1	0	3	2495

Csak a felső néhány sor a `head` paranccsal kérhető le (az alsó néhány sor pedig a `tail`-lel):

```
head( birthwt )
```

	low	age	lwt	race	smoke	ptl	ht	ui	ftv	bwt
85	0	19	182	2	0	0	0	1	0	2523
86	0	33	155	3	0	0	0	0	3	2551
87	0	20	105	1	1	0	0	0	1	2557

```

88  0  21 108    1    1  0  0  1  2 2594
89  0  18 107    1    1  0  0  1  0 2600
91  0  21 124    3    0  0  0  0  0 2622

```

Az oszlopok és a sorok is elnevezhetőek:

```
str( birthwt )
```

```

'data.frame':  189 obs. of  10 variables:
 $ low  : int  0 0 0 0 0 0 0 0 0 0 ...
 $ age  : int  19 33 20 21 18 21 22 17 29 26 ...
 $ lwt  : int  182 155 105 108 107 124 118 103 123 113 ...
 $ race : int  2 3 1 1 1 3 1 3 1 1 ...
 $ smoke: int  0 0 1 1 1 0 0 0 1 1 ...
 $ ptl  : int  0 0 0 0 0 0 0 0 0 0 ...
 $ ht   : int  0 0 0 0 0 0 0 0 0 0 ...
 $ ui   : int  1 0 0 1 1 0 0 0 0 0 ...
 $ ftv  : int  0 3 1 2 0 0 1 1 1 0 ...
 $ bwt  : int  2523 2551 2557 2594 2600 2622 2637 2637 2663 2665 ...

```

```
names( birthwt )
```

```

[1] "low"  "age"  "lwt"  "race" "smoke" "ptl"  "ht"   "ui"   "ftv"
[10] "bwt"

```

```
colnames( birthwt )
```

```

[1] "low"  "age"  "lwt"  "race" "smoke" "ptl"  "ht"   "ui"   "ftv"
[10] "bwt"

```

Az adatkeret a mátrixhoz hasonlóan indexelhető:

```
birthwt[ 3, ]
```

```

   low age lwt race smoke ptl ht ui ftv  bwt
87   0  20 105    1    1  0  0  0  1 2557

```

```
birthwt[ 3, 4 ]
```

```
[1] 1
```

```
birthwt[ 3, c( 5, 6 ) ]
```

```
      smoke ptl  
87      1     0
```

Sőt, ha vannak elnevezéseink, az is használható. A következő 4 mind egyenértékű:

```
birthwt[ , 10 ]
```

```
[1] 2523 2551 2557 2594 2600 2622 2637 2637 2663 2665 2722 2733 2751 2750 2769  
[16] 2769 2778 2782 2807 2821 2835 2835 2836 2863 2877 2877 2906 2920 2920 2920  
[31] 2920 2948 2948 2977 2977 2977 2977 2922 3005 3033 3042 3062 3062 3062 3062  
[46] 3062 3080 3090 3090 3090 3100 3104 3132 3147 3175 3175 3203 3203 3203 3225  
[61] 3225 3232 3232 3234 3260 3274 3274 3303 3317 3317 3317 3321 3331 3374 3374  
[76] 3402 3416 3430 3444 3459 3460 3473 3544 3487 3544 3572 3572 3586 3600 3614  
[91] 3614 3629 3629 3637 3643 3651 3651 3651 3651 3699 3728 3756 3770 3770 3770  
[106] 3790 3799 3827 3856 3860 3860 3884 3884 3912 3940 3941 3941 3969 3983 3997  
[121] 3997 4054 4054 4111 4153 4167 4174 4238 4593 4990 709 1021 1135 1330 1474  
[136] 1588 1588 1701 1729 1790 1818 1885 1893 1899 1928 1928 1928 1936 1970 2055  
[151] 2055 2082 2084 2084 2100 2125 2126 2187 2187 2211 2225 2240 2240 2282 2296  
[166] 2296 2301 2325 2353 2353 2367 2381 2381 2381 2410 2410 2410 2414 2424 2438  
[181] 2442 2450 2466 2466 2466 2495 2495 2495 2495
```

```
birthwt$bwt
```

```
[1] 2523 2551 2557 2594 2600 2622 2637 2637 2663 2665 2722 2733 2751 2750 2769  
[16] 2769 2778 2782 2807 2821 2835 2835 2836 2863 2877 2877 2906 2920 2920 2920  
[31] 2920 2948 2948 2977 2977 2977 2977 2922 3005 3033 3042 3062 3062 3062 3062  
[46] 3062 3080 3090 3090 3090 3100 3104 3132 3147 3175 3175 3203 3203 3203 3225  
[61] 3225 3232 3232 3234 3260 3274 3274 3303 3317 3317 3317 3321 3331 3374 3374  
[76] 3402 3416 3430 3444 3459 3460 3473 3544 3487 3544 3572 3572 3586 3600 3614  
[91] 3614 3629 3629 3637 3643 3651 3651 3651 3651 3699 3728 3756 3770 3770 3770  
[106] 3790 3799 3827 3856 3860 3860 3884 3884 3912 3940 3941 3941 3969 3983 3997  
[121] 3997 4054 4054 4111 4153 4167 4174 4238 4593 4990 709 1021 1135 1330 1474  
[136] 1588 1588 1701 1729 1790 1818 1885 1893 1899 1928 1928 1928 1936 1970 2055
```

```
[151] 2055 2082 2084 2084 2100 2125 2126 2187 2187 2211 2225 2240 2240 2282 2296
[166] 2296 2301 2325 2353 2353 2367 2381 2381 2381 2410 2410 2410 2414 2424 2438
[181] 2442 2450 2466 2466 2466 2495 2495 2495 2495
```

```
birthwt[ , "bwt" ]
```

```
[1] 2523 2551 2557 2594 2600 2622 2637 2637 2663 2665 2722 2733 2751 2750 2769
[16] 2769 2778 2782 2807 2821 2835 2835 2836 2863 2877 2877 2906 2920 2920 2920
[31] 2920 2948 2948 2977 2977 2977 2977 2922 3005 3033 3042 3062 3062 3062 3062
[46] 3062 3080 3090 3090 3090 3100 3104 3132 3147 3175 3175 3203 3203 3203 3225
[61] 3225 3232 3232 3234 3260 3274 3274 3303 3317 3317 3317 3321 3331 3374 3374
[76] 3402 3416 3430 3444 3459 3460 3473 3544 3487 3544 3572 3572 3586 3600 3614
[91] 3614 3629 3629 3637 3643 3651 3651 3651 3651 3699 3728 3756 3770 3770 3770
[106] 3790 3799 3827 3856 3860 3860 3884 3884 3912 3940 3941 3941 3969 3983 3997
[121] 3997 4054 4054 4111 4153 4167 4174 4238 4593 4990 709 1021 1135 1330 1474
[136] 1588 1588 1701 1729 1790 1818 1885 1893 1899 1928 1928 1928 1936 1970 2055
[151] 2055 2082 2084 2084 2100 2125 2126 2187 2187 2211 2225 2240 2240 2282 2296
[166] 2296 2301 2325 2353 2353 2367 2381 2381 2381 2410 2410 2410 2414 2424 2438
[181] 2442 2450 2466 2466 2466 2495 2495 2495 2495
```

```
birthwt[[ "bwt" ]]
```

```
[1] 2523 2551 2557 2594 2600 2622 2637 2637 2663 2665 2722 2733 2751 2750 2769
[16] 2769 2778 2782 2807 2821 2835 2835 2836 2863 2877 2877 2906 2920 2920 2920
[31] 2920 2948 2948 2977 2977 2977 2977 2922 3005 3033 3042 3062 3062 3062 3062
[46] 3062 3080 3090 3090 3090 3100 3104 3132 3147 3175 3175 3203 3203 3203 3225
[61] 3225 3232 3232 3234 3260 3274 3274 3303 3317 3317 3317 3321 3331 3374 3374
[76] 3402 3416 3430 3444 3459 3460 3473 3544 3487 3544 3572 3572 3586 3600 3614
[91] 3614 3629 3629 3637 3643 3651 3651 3651 3651 3699 3728 3756 3770 3770 3770
[106] 3790 3799 3827 3856 3860 3860 3884 3884 3912 3940 3941 3941 3969 3983 3997
[121] 3997 4054 4054 4111 4153 4167 4174 4238 4593 4990 709 1021 1135 1330 1474
[136] 1588 1588 1701 1729 1790 1818 1885 1893 1899 1928 1928 1928 1936 1970 2055
[151] 2055 2082 2084 2084 2100 2125 2126 2187 2187 2211 2225 2240 2240 2282 2296
[166] 2296 2301 2325 2353 2353 2367 2381 2381 2381 2410 2410 2410 2414 2424 2438
[181] 2442 2450 2466 2466 2466 2495 2495 2495 2495
```

A nem dupla szögletes zárójellel történő indexelés eltérése, hogy nem a kiválasztott vektort, hanem egy csak a kiválasztott vektorból álló data frame-et ad vissza:

```
birthwt[[ "bwt" ]]
```

```
[1] 2523 2551 2557 2594 2600 2622 2637 2637 2663 2665 2722 2733 2751 2750 2769
[16] 2769 2778 2782 2807 2821 2835 2835 2836 2863 2877 2877 2906 2920 2920 2920
[31] 2920 2948 2948 2977 2977 2977 2977 2922 3005 3033 3042 3062 3062 3062 3062
[46] 3062 3080 3090 3090 3090 3100 3104 3132 3147 3175 3175 3203 3203 3203 3225
[61] 3225 3232 3232 3234 3260 3274 3274 3303 3317 3317 3317 3321 3331 3374 3374
[76] 3402 3416 3430 3444 3459 3460 3473 3544 3487 3544 3572 3572 3586 3600 3614
[91] 3614 3629 3629 3637 3643 3651 3651 3651 3651 3699 3728 3756 3770 3770 3770
[106] 3790 3799 3827 3856 3860 3860 3884 3884 3912 3940 3941 3941 3969 3983 3997
[121] 3997 4054 4054 4111 4153 4167 4174 4238 4593 4990 709 1021 1135 1330 1474
[136] 1588 1588 1701 1729 1790 1818 1885 1893 1899 1928 1928 1928 1936 1970 2055
[151] 2055 2082 2084 2084 2100 2125 2126 2187 2187 2211 2225 2240 2240 2282 2296
[166] 2296 2301 2325 2353 2353 2367 2381 2381 2381 2410 2410 2410 2414 2424 2438
[181] 2442 2450 2466 2466 2466 2495 2495 2495 2495
```

```
str( birthwt[[ "bwt" ] ] )
```

```
int [1:189] 2523 2551 2557 2594 2600 2622 2637 2637 2663 2665 ...
```

```
head( birthwt[ "bwt" ] )
```

```
      bwt
85 2523
86 2551
87 2557
88 2594
89 2600
91 2622
```

```
str( birthwt[ "bwt" ] )
```

```
'data.frame':  189 obs. of  1 variable:
 $ bwt: int  2523 2551 2557 2594 2600 2622 2637 2637 2663 2665 ...
```

Használhatunk különféle módszereket (az alábbiak közül a második a logikai indexelés miatt fog működni):


```
head( birthwt[ , c( "lwt", "smoke" ) ] )
```

```
      lwt smoke
85 182      0
86 155      0
87 105      1
88 108      1
89 107      1
91 124      0
```

```
head( birthwt[ birthwt$smoke==1, ] )
```

```
      low age lwt race smoke ptl ht ui ftv  bwt
87      0  20 105     1     1  0  0  0   1 2557
88      0  21 108     1     1  0  0  1   2 2594
89      0  18 107     1     1  0  0  1   0 2600
94      0  29 123     1     1  0  0  0   1 2663
95      0  26 113     1     1  0  0  0   0 2665
100     0  18 100     1     1  0  0  0   0 2769
```

```
head( birthwt[ birthwt$smoke==1&birthwt$race==1, ] )
```

```
      low age lwt race smoke ptl ht ui ftv  bwt
87      0  20 105     1     1  0  0  0   1 2557
88      0  21 108     1     1  0  0  1   2 2594
89      0  18 107     1     1  0  0  1   0 2600
94      0  29 123     1     1  0  0  0   1 2663
95      0  26 113     1     1  0  0  0   0 2665
100     0  18 100     1     1  0  0  0   0 2769
```

Az adatkeret heterogén:

```
birthwt$nev <- "a"
head( birthwt )
```

```
      low age lwt race smoke ptl ht ui ftv  bwt nev
85      0  19 182     2     0  0  0  1   0 2523   a
86      0  33 155     3     0  0  0  0   3 2551   a
87      0  20 105     1     1  0  0  0   1 2557   a
```

```

88  0  21 108    1    1  0  0  1  2 2594  a
89  0  18 107    1    1  0  0  1  0 2600  a
91  0  21 124    3    0  0  0  0  0 2622  a

```

```
str( birthwt )
```

```

'data.frame':  189 obs. of  11 variables:
 $ low  : int  0 0 0 0 0 0 0 0 0 0 ...
 $ age  : int  19 33 20 21 18 21 22 17 29 26 ...
 $ lwt  : int  182 155 105 108 107 124 118 103 123 113 ...
 $ race : int  2 3 1 1 1 3 1 3 1 1 ...
 $ smoke: int  0 0 1 1 1 0 0 0 1 1 ...
 $ ptl  : int  0 0 0 0 0 0 0 0 0 0 ...
 $ ht   : int  0 0 0 0 0 0 0 0 0 0 ...
 $ ui   : int  1 0 0 1 1 0 0 0 0 0 ...
 $ ftv  : int  0 3 1 2 0 0 1 1 1 0 ...
 $ bwt  : int  2523 2551 2557 2594 2600 2622 2637 2637 2663 2665 ...
 $ nev  : chr  "a" "a" "a" "a" ...

```

2.3.5. Lista

A lista heterogén, egydimenziós adatszerkezet.

Legegyszerűbben elemei felsorolásával hozható létre, a `list` függvényt használva:

```
lista <- list( sz = szamvektor, k = karaktervektor, m = szammatrix, df = birthwt[ 1:5, ] )
lista
```

```

$sz
    egy ketto harom
      4      1    99    NA    NA    NA    NA    NA    NA    999

```

```

$k
[1] "a"  "b"  "xyz"

```

```

$m
      [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

```

```
$df
  low age lwt race smoke ptl ht ui ftv  bwt nev
85   0  19 182   2     0   0  0  1   0 2523   a
86   0  33 155   3     0   0  0  0   3 2551   a
87   0  20 105   1     1   0  0  0   1 2557   a
88   0  21 108   1     1   0  0  1   2 2594   a
89   0  18 107   1     1   0  0  1   0 2600   a
```

```
str( lista )
```

List of 4

```
$ sz: Named num [1:10] 4 1 99 NA NA NA NA NA NA 999
..- attr(*, "names")= chr [1:10] "egy" "ketto" "harom" "" ...
$ k : chr [1:3] "a" "b" "xyz"
$ m : int [1:3, 1:2] 1 2 3 4 5 6
$ df:'data.frame': 5 obs. of  11 variables:
..$ low   : int [1:5] 0 0 0 0 0
..$ age   : int [1:5] 19 33 20 21 18
..$ lwt   : int [1:5] 182 155 105 108 107
..$ race  : int [1:5] 2 3 1 1 1
..$ smoke: int [1:5] 0 0 1 1 1
..$ ptl   : int [1:5] 0 0 0 0 0
..$ ht    : int [1:5] 0 0 0 0 0
..$ ui    : int [1:5] 1 0 0 1 1
..$ ftv   : int [1:5] 0 3 1 2 0
..$ bwt   : int [1:5] 2523 2551 2557 2594 2600
..$ nev   : chr [1:5] "a" "a" "a" "a" ...
```

Számmal és – ha van neki – névvel is indexelhető:

```
lista[[ 1 ]]
```

```
egy ketto harom
 4      1      99      NA      NA      NA      NA      NA      NA      999
```

```
lista$sz
```

```
egy ketto harom
 4      1      99      NA      NA      NA      NA      NA      NA      999
```

```
lista[[ "sz" ]]
```

```
    egy ketto  három
    4      1    99   NA    NA    NA    NA    NA    NA   999
```

Az egy zárójellel történő indexelés látszólag ugyanaz, de csak látszólag:

```
lista[ 1 ]
```

```
$sz
    egy ketto  három
    4      1    99   NA    NA    NA    NA    NA    NA   999
```

```
typeof( lista[[ 1 ] ] )
```

```
[1] "double"
```

```
typeof( lista[ 1 ] )
```

```
[1] "list"
```

Tartomány is indexelhető:

```
lista[ 1:2 ]
```

```
$sz
    egy ketto  három
    4      1    99   NA    NA    NA    NA    NA    NA   999
```

```
$k
[1] "a"    "b"    "xyz"
```

```
lista[[ 1:2 ]]
```

```
[1] 1
```

Az előbbi dolgok természetesen kombinálhatóak is:

```
idx <- "sz"  
lista[[ idx ]]
```

```
egy ketto harm  
  4      1    99    NA    NA    NA    NA    NA    999
```

Az adatkeret igazából egy, az oszlopokból - mint vektorokból - összerakott lista (tehát két szűkítés van: az elemek csak vektorok lehetnek és ugyanolyan hosszúaknak kell lenniük).

3 Függvények

Sok esetben előfordul, hogy egy statisztikai programban ismétlődő részek vannak. Tekintsük például a következő kódot (`x` egy valós számokat tartalmazó vektor):

```
x <- rnorm(10)
x
```

```
[1] 0.33571523 0.13614490 -1.06994854 -0.76614437 0.05255738 0.41045994
[7] 0.85918586 -0.31820200 0.02077204 -1.06996005
```

```
n <- length(x)
osszeg <- sum(x)
atlag <- osszeg/n
sum((x - atlag)^2)/(n - 1)
```

```
[1] 0.4244777
```

Valószínűleg sokan felismerik, hogy ezzel az `x`-ben lévő elemek varianciáját számoltuk ki. Erre ezerszer szükség lehet egy statisztikai elemzésben: egyszer kiszámolhatjuk a testtömegek varianciáját, aztán a testmagasságokét, kellhet varianciát számolni, ha valamilyen regressziót futtatunk, ha főkomponens-elemzést hajtunk végre, egyszóval milliónyi feladatnál, akár ugyanazon szkripten belül is, pláne a különböző szkriptekben.

Megtehetnénk természetesen, hogy minden ilyen esetben elhelyezzük ugyanazt a kódrészletet: kimásoljuk a fentit, és mindenhová beillesztjük, `x`-et a megfelelő változónévre cserélve – de remélhetőleg zsigerből érezhető, hogy ez nagyon-nagyon szerencsétlen megoldás lenne. Azon túl, hogy nagyon hosszúvá tenné a kódot (ami nem a gépnek probléma, hanem nekünk, olvashatóság szempontjából), azon túl, hogy megnehezíti a kód elemzését, értelmezését, mert nem lesz gyorsan és egyértelműen beazonosítható, hogy mely helyeken történik meg ez a számolás, a fő probléma, hogy borzasztóan megnehezítené a kód karbantartását, a hibajavításokat, fejlesztéseket. Képzeljük el például, hogy miután végeztünk a fenti bemásolgatós megoldással, észrevesszük, hogy az `osszeg` változót teljesen felesleges volt elmenteni, mert csak egyetlen egyszer hivatkozunk rá, nyugodtan beírhattuk volna közvetlenül a következő sorba. Mit csinálunk? Ha a bemásolgatós megoldással éltünk, akkor nincs más lehetőség, mind az ezer helyen át kell írni...!

Mennyivel jobb lenne ezt a kódot csak egyetlen egy helyen megírni, adni neki egy nevet, mondjuk **variancia**, és minden alkalommal, amikor szükség van rá a programunkban, azt írni ott, hogy „most futtasd le a **variancia**-t”. Egyetlen rövid utasítás, ami azt jelentené, hogy innentől ugorj át erre az – egyetlen helyen megírt – kódra, futtasd le, ami ott van, majd ha végeztél, térj vissza pontosan oda, ahol mondtuk, hogy futtasd le ezt a kód. Az ilyen kódot hívjuk úgy a programozásban, hogy függvény, az előbbi folyamatra pedig azt mondjuk, hogy meghívtuk a függvényt. Mindezt akárhány alkalommal megtehetjük, mindig ugyanaz az – egyetlen helyen megírt – függvény fut le, és a végén mindig oda ugrik vissza, ahonnan éppen hívtuk. A fenti kódot, a függvény „tartalmát” a függvény törzsének szoktuk mondani.

Ezzel a módszerrel sokkal olvashatóbb lesz a kód, azonnal és egyértelműen látszódni fog, hogy hol használtuk ezt a variancia-számítást, de a legfontosabb, hogy ha valamit módosítanunk kell a variancia-számításban, akkor ezt elég egy helyen megtenni, és ettől automatikusan az összes hívásnál meg fog változni.

3.1. Paraméterátadás és visszatérési érték

Egyetlen egy dolgot kell megoldani, az információk átadását. Mindkét irányban: valahogy el kell juttatnunk a függvényhez, hogy mi az a vektor, aminek a varianciáját számolni akarjuk, illetve fordítva, valahogy vissza kell juttatnunk a kiszámított varianciát a programhoz.

Az előbbi megoldás kinézetét már sugallja a fenti kód is: a függvényben meghatározunk egy változónevet (**x**), és azon hajtjuk végre a műveleteket a függvény törzsében, az információátadást pedig úgy valósítjuk meg, hogy a függvény hívásakor megadjuk, hogy milyen konkrét érték, például programbeli változó – mondjuk **testtomegek** – kerüljön át a függvénybe ezen a néven. Az ilyen **x**-et szép néven paraméternek, a konkrét értéket, amit a függvény hívásakor megadunk, argumentumnak nevezzük¹; az egész mechanizmus neve pedig paraméterátadás. Lényegében az történik, hogy **x** szerepét a **testtomegek** változó fogja játszani – ennél a hívásnál. De később, ha meghívjuk a függvényt a **testmagassagok** argumentummal, akkor az fogja **x** szerepét játszani. Nekünk a függvényben azonban általánosan kellett meghatározni, hogy mit kell tenni, erre szolgál az **x**, azon írjuk le, hogy mi *általában* az eljárás, arról pedig az R gondoskodik, hogy ez mindig a megfelelő értéken hajtódjon végre. A háttérben az történik, hogy amikor meghívjuk a függvényt a **testtomegek** változóval mint argumentummal, akkor az R készít egy másolatot a **testtomegek**-ről, elnevezi **x**-nek, és lefuttatja rajta a függvény törzsét. A „készít róla másolatot” kitétel fontos, ez ugyanis azt is jelenti, hogy a függvény törzse bármit is csinál az **x**-szel, az nem fog visszahatni a hívó programban található **testtomegek** változóra: ha a függvény át is írja **x**-et, attól még a **testtomegek** a hívó programban ugyanaz marad (hiszen ilyenkor a függvény csak a **testtomegek** egy *másolatát* módosította, aminek semmi

¹Az R a paramétert formális argumentumnak hívja; ezt a szóhasználatot egyébként a magyar szaknyelv is megengedi mint a paraméter szinonimája. De sokszor nem vesződnek ezzel a különbségtétellel, és egyszerűen argumentumot mondanak.

köze a **testtomegek**-hez!). Ezt szokták érték szerinti paraméterátadásnak nevezni². Egyetlen megjegyzés a végére: ne ijedjünk meg a „készít róla másolatot” megfogalmazástól. Ezt csak az elmagyarázást segítő szófordulat volt a részemről, az R ezt igyekszik ügyesen kezelni, és nem feltétlenül készít³ ténylegesen másolatot, azaz a dolog nem jelenti automatikusan azt, hogy minden függvényhívásnál teljesítmény-veszteség keletkezik a másolgatások miatt, aminek van valamennyi időigénye.

És mi a helyzet fordított irányban? Hogyan adhatunk vissza információt a hívó programnak? Az első dolog, amit tudni kell, hogy R-ben egy függvény csak *egyetlen* értéket adhat vissza⁴; ezt hívjuk visszatérési értéknek. Ez elsőre rosszul hangozhat, de ne ijedjünk meg, ez semmilyen érdemi limitációt nem jelent: a visszatérési érték ugyanis nyugodtan lehet egy lista is! Abba pedig akármilyen objektumból akármennyit bepakolhatunk. A visszatérési értéket megadni a **return**-nel lehet, a fenti példában azt kellene írunk az utolsó sorban, hogy **return**(**sum**((**x** - **atlag**)²)/(**n** - 1)). Ez azt jelenti, hogy a függvény végrehajtása befejeződik, a vezérlés visszakerül a hívó programhoz, olyan módon, hogy a függvényhívás behelyettesítődik a visszatérési értékkel. Ez utóbbi nagyon fontos: a dolgot úgy kell elképzelnünk, mintha a meghívott függvény helyébe odaírnánk a visszatérési értékét, és úgy menne tovább a végrehajtás. Két technikai megjegyzés a végére. Az egyik, hogy nem kötelező, hogy a **return** legyen egy függvény utolsó utasítása, de mivel a **return** elérésekor a vezérlés visszakerül a hívóhoz, így túl sok értelme sincs bármit utána írni. Jellemzően akkor fordul elő, hogy **return**-t látunk egy függvény közepén, ha valójában egy feltételes részben van, tehát nem biztos, hogy végrehajtott: a **return** megvalósítja azt, hogy *ha* oda kerül a végrehajtás, akkor visszatér (és így a függvény többi része végre sem hajtott), de ha nem, akkor a **return** le sem fut, és így mehet tovább a függvényben a végrehajtás. A másik megjegyzést talán még gyakrabban használjuk a gyakorlatban, ez pedig az a szabály, hogy ha a függvényben egyáltalán nincs **return**, akkor a visszatérési érték az utolsóként kiértékelt kifejezés értéke lesz. Ez a gyakorlatban azért fontos, mert elég tipikus, hogy egy függvény utolsó utasítása egyszerűen egy változó neve, vagy valamilyen egyszerű művelet egy változóval – ekkor ez az érték fog automatikusan **return**-ölni. A fenti példában tehát igazából elég lenne a függvény utolsó sorában annyit írni, hogy **sum**((**x** - **atlag**)²)/(**n** - 1), a **return** nem is kell, az eredmény ugyanaz lesz.

Ha egy függvény bármilyen más hatást vált ki azon kívül, hogy kiszámolja a visszatérési értéket és azt visszaadja, akkor azt mellékhatásnak nevezzük, az ilyen függvényt pedig mellékhatásos

²Az alternatívája a cím szerinti paraméterátadás, ami nagyon leegyszerűsítve az, ha a függvény nem a változó egy másolatát kapja meg, hanem azt a memóriacímet, ahol a változó van. Ez más, mert bár kiolvasva a memóriacímet megkapjuk a változó tartalmát, ugyanúgy mint az érték szerinti átadásnál, viszont a memóriacím ismeretében akár módosíthatjuk is azt: ha valamit átírunk ott, az módosítja a hívó program számára is a változót. Cím szerinti átadáshoz hasonló mechanizmus megvalósítható R-ben is, de külön erőfeszítést igényel, az alapvető működési mód az R-ben az érték szerinti átadás. A valóságban az érték és cím szerinti átadásnak sok egyéb finomsága van, más paraméterátadási módok is léteznek, de ezek számunkra nem lesznek most fontosak.

³Ezt hívják copy-on-modify mechanizmusnak, ez szabályozza, hogy mikor készül másolat, és ha nem készül, akkor hogyan oldja meg a paraméterátadást.

⁴És az érték szerinti paraméterátadás miatt argumentumot sem használhatunk információ visszajuttatására, ahogy az más nyelvben előfordulhatna.

függvénynek. Ez elsőre meglepő lehet (mégis mi más hatása lehet egy függvénynek azon kívül, hogy kiszámolja a visszatérési értéket és azt visszaadja?!), de ha meggondoljuk, valójában egészen banális dolgok tartoznak ide – kezdjük például kapásból a kiíratással! Az ugyanis teljes mértékben egy mellékhatás: ha egy függvény hatására valami megjelenik a konzolon, annak nyilván semmi köze a visszatérési értékhez. De az is mellékhatás, ha egy fájlt kiírunk a merevlemezre, vagy mondjuk ha megjelenítünk egy ábrát. Az R programozási filozófiája miatt alapvetően kerülni célszerű a mellékhatásokat⁵, de mint az előbbi példák mutatják, bizonyos esetben elkerülhetetlenek.

3.2. Saját függvények létrehozása

A fentiekkel szinte már mindent tudunk ahhoz, hogy saját függvényt tudjunk definiálni. Amint láttuk, ez fontos: ez minden alkalommal jól jön, ha ismétlődő kódrészletek vannak a statisztikai számításunkban. Nézzük, hogyan tehetjük ezt meg egész pontosan.

Először is, függvényt ugyanúgy értékadással hozhatunk létre, ahogy létrehozunk egy – korábban még nem létező – változót. Amit a változónak értékül kell adnunk, az a `function` kulcsszó, ennek hatására a létrejövő változó egy függvény lesz, amit később meghívhatunk. A `function` után gömbölyű zárójel következik, benne a paraméterek – ennek részleteit a következő pontban fogjuk látni – utána pedig a függvény törzse. Ez egyetlen R kifejezés lehet, ami persze az esetek túlnyomó többségében értelmetlenül kevés, de semmi gond: erre szolgál a blokk-képzés. A blokk-képzés azt jelenti, hogy fogunk utasításokat, és összefogjuk őket egyetlen egységbe – ez lesz a blokk – amik úgy viselkednek, mintha az egyetlen kifejezés lenne. (A végrehajtása természetesen azt jelenti, hogy mindegyik benne lévő utasítás végrehajtódik egymás után, ahogy bármilyen R kódnál történne.) A blokk-képzés jele az R-ben a kapcsos zárójel, ezzel lehet több utasítást egyetlen blokkba összefogni.

Mindezek alapján egy saját függvény létrehozása a következőképp néz ki a fenti példában:

```
variancia <- function(x) {  
  n <- length(x)  
  osszeg <- sum(x)  
  atlag <- osszeg/n  
  sum((x - atlag)^2)/(n - 1)  
}
```

Ezután bármikor meghívhatjuk:

⁵Ezen a ponton visszatérhetünk egy pillanatra arra a korábbi megállapításra, hogy egy függvény lecserélhető a visszatérési értékére. Ha ez megvalósul, akkor azt mondjuk, hogy fennáll a hivatkozási átlátszóság; ennek a feltétele, hogy a függvény mellékhatásmentes legyen, és adott argumentumokra mindig, determinisztikusan ugyanazt a visszatérési értéket szolgáltatassa. Az ilyeneket tiszta függvénynek szokták hívni a programozásméletben.

```
x <- rnorm(10)
variancia(x)
```

```
[1] 1.016883
```

(Természetesen erre a célra van beépített – és a fenténél sokkal gyorsabb – függvény, a neve `var`, de ez most csak az illusztráció célját szolgálta.)

Ami feltűnhet a fentiben, hogy sem a paraméterek, sem a visszatérési érték típusát nem kellett megadni. Ez következik abból, hogy az R dinamikus típusrendszerű nyelv, és minden igaz rá, amit a dinamikus típusrendszerről az adattípusoknál elmondtunk: egyszerűsíti az életet (nem kell vacakolni a típusok felsorolásával, nyugodtan előfordulhat, hogy az argumentumok fényében nem mindig ugyanolyan típusú eredményt adunk vissza), de cserében megszüntet egy védelmi vonalat, ami kinyit hibalehetőségeket. Nem csak arról van szó, hogy a fenti függvényt meghívhatjuk például egyelemű vektorral – ami az `n-1`-gyel való osztás miatt lesz elég nagy baj – de az előzőekben mondtak miatt még az ellen sem lesz beépített védelmünk, hogy valaki mondjuk egy szöveg varianciáját álljon neki kiszámíttatni...! Az R-ben minden ilyenre nekünk kell figyelni. És érdemes is: ilyen esetekben fontos lehet, hogy mielőtt bármit csinálunk, ellenőrizzük a kapott argumentumot vagy argumentumokat; erre vannak bevált eszközök és módszerek⁶ az R-ben. Ez különösen akkor helyes gyakorlat, ha a függvényt nem csak mi magunk fogjuk meghívni – ekkor még mondhatnánk, hogy a meghívó oldaláról tudjuk biztosítani a megfelelő argumentumokat, bár az igazság az, hogy ilyenkor is sokszor jobb a biztonság – hanem mások is meghívhatják, például, mert egy R csomagba kerül ki.

Az R programozásáról szóló fejezetben látni fogjuk, hogy lesznek olyan függvények amelyeknek az egyik argumentuma maga is függvény. Ez lehet egy fentihez hasonló függvény: létrehozunk egy függvényt, nevet adunk neki, majd ezt a nevet adjuk meg ilyenkor argumentumként. Ilyen esetekben azonban elég gyakori, hogy a függvényt nem akarjuk külön elmenteni, saját néven, ugyanis máshol nincsen rá szükség, és csak ott akarnánk, direkte a kódba beleírni a függvényt az argumentum helyébe, anélkül, hogy előtte külön néven elmentjük. (Lényegében arról van szó, hogy szeretnénk megtenni ugyanazt a függvényekkel, amit megtehetünk más kifejezésekkel. Hiszen ha például gyököt akarunk vonni $3 + 1$ -ből, akkor egyáltalán nem kötelező előbb bementeni egy külön, névvel rendelkező változóba a $3 + 1$ -et, majd arra a változóra meghívni a `sqrt`-ot, egész nyugodtan megtehetjük azt is, hogy azt írjuk, hogy `sqrt(3 + 1)`.) A helyzet az, hogy ez minden további nélkül működik függvényekkel is! A fenti szintaktikával kell létrehozni függvényt, de egyáltalán nem kötelező, hogy azt bármibe belenyilazzuk. Az így létrehozott függvény nyugodtan beírható az argumentum helyébe, a dolog tökéletesen működőképes lesz. Az ilyen függvényt hívjuk szép szóval anonim függvénynek.

Egyetlen megjegyzés a végére. Az R 4.1.0-s verziója bevezetett egy rövidítést: a `\(x)` pontosan ugyanaz mint a `function(x)`. Ezt inkább anonim függvényeknél szokták használni, hiszen ott

⁶Legegyszerűbb esetben használhatjuk a `stopifnot` függvényt, de van külön csomag is erre a célra, például az `assertthat` vagy a `checkmate`.

jobban mutat, ha kevesebbet kell írni, mivel ott a függvény egy másik függvény argumentumába van beírva, ami elég zsúfolt kódot eredményezhet már eleve is.

3.3. R csomagok használata

Volt már róla szó, hogy az R erejét nem kis részt az adja, hogy hihetetlen mennyiségű csomag érhető el hozzá, melyek a legkülönbébb statisztikai problémák megoldását teszik lehetővé. Most már pontosabb definícióját adhatjuk annak, hogy mi az egyáltalán, hogy R csomag: tartalmilag összetartozó függvények gyűjteménye (dokumentációval és esetleg adatokkal kiegészítve). Nézzük meg részletesebben, hogy miről van szó!

Amikor azt mondtam, hogy „statisztikai problémák megoldása”, akkor azt lényegében így értettem: megfelelő függvények megírása. Hiszen ezeket a statisztikai problémákat éppen úgy oldjuk meg, hogy létrehozunk (egy vagy több) függvényt! A megoldás abban nyilvánul meg, hogy létrehozunk függvényt vagy függvényeket, amik a szükséges számításokat elvégzik, tehát tömörítik a statisztikai tudásunkat azáltal, hogy a feladatot – vagy valamilyen részfeladatot – megoldják, ahogy a fenti függvény megoldja a variancia kiszámításának a feladatát. Azzal a nem elhanyagolható eltéréssel, hogy a függvény természetesen nem biztos, hogy 4 sor, lehet éppenséggel 4 ezer is. Elég tipikus, hogy egy adott témakör számításai több függvényt igényelnek, ezeket dokumentációval kell ellátni, néha érdemes adatbázisokat is mellékelni – az R csomag nem más, mint ilyenek gyűjteménye. Amikor betöltünk egy R csomagot, akkor elérhetővé válnak a benne definiált függvények, vagyis azáltal „férünk hozzá az erejéhez”, hogy meg tudjuk hívni ezeket a függvényeket, így egyetlen sorban elérünk néha több ezer sornyi, adott esetben igen komplex statisztikai számítást, amit más, jó esetben a témában jártas szakértők megírtak számunkra, így mi készen használhatjuk.

Kis kitérő: eddig még nem töltöttünk be semmilyen csomagot, és mégis voltak elérhető függvényeink, mint az `rnorm` vagy a épp az `str`. Eddig ezt úgy kezeltem, mintha ezek valamiféle „beépített” függvények lennének, amikhez nem kell csomagot betölteni... csak hogy ez nem igaz. Ezek szintén csomagban vannak, ugyanúgy mint bármely más függvény (általában is, ha megnézzük egy függvény sűrűjét, akkor a bal felső sarokban, kapcsos zárójelben látjuk, hogy melyik csomagban, innen kiderül, hogy az `rnorm` a `stats`-ban, az `str` a `utils`-ban van), és ezeket a csomagokat nagyon is be kell tölteni, hogy elérjük a függvényeiket – csak épp az R ezt megteszi helyettünk! Van ugyanis hét csomag⁷ amit az R az indulásakor *automatikusan* betölt, így a függvényei elérhetővé válnak úgy, hogy látszólag nem kellett semmit sem tennünk – de a valóságban ez is ugyanúgy csomagbetöltés volt.

⁷A `base`, a `stats`, a `methods`, a `graphics`, a `grDevices`, a `utils` és a `datasets`. Egész pontosan az a mechanizmus, hogy a `base` csomag mindenképp betöltődik induláskor, utána pedig azok töltődnek még be ezen felül automatikusan, amik a `defaultPackages` nevű opcióban szereplnek; az előbbi lista ennek az alapértelmezését tükrözi.

Még mielőtt a csomagok betöltésére térünk, egy dolgot meg kell beszélni: a csomagok telepítését, betölteni ugyanis csak telepített csomagot⁸ tudunk. Bárki készíthet R csomagot, egy R csomag lehet egyetlen összecsomagolt zip-fájl, de van egy eszköz, ami miatt mégsem alakul ki káosz ebből a dologból, azaz nem zip-fájlok keringenek össze-vissza, ez az eszköz pedig nem más, mint a CRAN (Comprehensive R Archive Network). Ez az R csomagok egy központi repozitóriuma, ahol szinte minden fontos R csomag megtalálható, így ez kiküszöböli a káoszt: szinte mindenki erre az egy helyre tölti fel az R csomagját, ha készít ilyet, és szinte mindenki erről az egy helyről tölti le az elkészített R csomagokat. (A „szinte” kitétel annak szól, hogy léteznek más repozitóriumok is, megfelelő csomag segítségével lehet Github-repozitóriumból is installálni csomagot, illetve elvileg lehet tényleg zip-fájlból is telepíteni csomagot, de ezek jelentősége eltörpül a CRAN mellett.) A CRAN ráadásul ellenőrzi is a csomagot, egyébként elég szigorúan, tehát ez egyfajta minőségbiztosítás is. (Mindazonáltal az ellenőrzés formai, tehát a csomag szintaktikai helyességére vonatkozik, nem a statisztikai – vagy bármilyen más – tartalmának tudomány helyességére!) A CRAN-ról történő telepítéshez az R külön utasítást tartalmaz, így valójában nem kell semmilyen zip-fájl letöltéssel vesződnünk. Ha RStudio-t használunk, akkor választhatjuk egyszerűen a Tools menü Install packages pontját, itt alapértelmezett a CRAN-ról történő telepítés, így csak a csomag nevét – vagy több csomag esetén a neveiket, vesszővel elválasztva – kell megadni, ebben automatikus kiegészítés segít is, majd az Install gombra kattintva megtörténik a telepítés. A konzolon láthatjuk, hogy ez igazából egyenértékű az `install.packages` függvény meghívásával, argumentumként megadva a csomag nevét; választhatjuk ezt a megoldást is. (Viszont ezt nem illik magába a szkriptbe beleírni, mert ez azt jelentené, hogy ha az egész szkriptet egyben futtatjuk, akkor az minden alkalommal nekiáll telepíteni a csomagot.) Ha egyszer megtörtént a telepítés⁹, onnantól már betölthetjük a csomagot.

Hogy legyen egy konkrét példánk csomag betöltéséhez, tekintsük azt a feladatot, hogy szeretnénk 10 véletlenszámot generálni, de ezúttal egy standard inverz exponenciálisnak nevezett eloszlásból. Hogy mi ez az eloszlás, az most mindegy is, ami viszont fontos, hogy – szemben a normálissal – az R erre nem tartalmaz „beépített” függvényt. De semmi vész, szerencsére az `actuar` nevű csomagban van egy függvény, a neve `rinvexp`, ami pont ezt valósítja meg! Első pont: ha korábban nem tettünk meg, telepítenünk kell az `actuar` csomagot; e nélkül nem tudunk továbbhaladni. Második pont: ha telepítettük, akkor be kell tölteni.

Ezen a ponton kettéágaznak a lehetőségek. Még mielőtt belevágunk, elsőként állapítsuk meg, hogy betöltés nélkül nem tudjuk lefuttatni a függvényt, hibát fog adni:

⁸Az R szokásos installációja a korábban említett, induláskor automatikusan betöltődő csomagokon kívül még kb. egy tucatnyi csomagot telepít, tehát ezeket be kell ugyan kézzel tölteni, de külön telepíteni nem kell. Ezen kívül minden más csomagot először telepíteni kell.

⁹Egyetlen megjegyzés ehhez. A CRAN-en a csomagok fent vannak forráskóddal, illetve Windows-ra előre lefordított (binary) verzióban is. Ez azt jelenti, hogy Linux-on a CRAN-ról történő telepítés mindenképp a forráskód fordítását fogja jelenteni, Windows-on viszont nem feltétlenül. Néha azonban az a helyzet, hogy a forráskódú verzió frissebb, mint a lefordított; ilyenkor a Windows-on is jól jöhet a forráskódból fordítás. Ehhez azonban fordítóeszköz fog kelleni! Szerencsére ehhez van kész eszköztár az R-hez, a neve RTools¹⁰, érdemes telepíteni.

```
rinvexp(10)
```

```
Error in rinvexp(10): could not find function "rinvexp"
```

Az első lehetőség, hogy a `::` (két kettőspont, néha szokták magyarul négyespontnak is hívni) operátort használjuk: először leírjuk a csomag nevét, utána négyespontot teszünk, majd a függvény nevét a csomagból. Ez így már lefut:

```
actuar::rinvexp(10)
```

```
[1] 0.4863114  4.5853484 33.4943526  0.5960459  1.8545105  0.6682524  
[7] 0.6944344 13.9006727  0.5690046  0.3799980
```

Magyarra lefordítva az `actuar::rinvexp` praktikusán azt jelenti, hogy „az `actuar` csomag `rinvexp` nevű függvénye”.

Ez nem szó szerint a csomag betöltése, abban az értelemben, hogy ettől nem válik a csomag összes többi függvénye elérhetővé, csak ez az egy (és ez az egy is csak ekkor, ha később újra szükségünk van erre a függvényre, akkor megint elé kell írni, hogy `actuar::`). A csomag szó szerinti betöltését a `library` függvénnyel¹¹ hajthatjuk végre:

```
library(actuar)
```

Kapcsolódás csomaghoz: `'actuar'`

The following objects are masked from 'package:stats':

```
sd, var
```

¹¹Ennek van egy alternatívája, a `require` függvény. A kettő nagyon hasonlít egymásra, az egyetlen különbség akkor bukik ki, ha nem telepített könyvtárat akarunk betölteni: ekkor a `library` hibát ad, a `require` csak figyelmeztetést, és visszatérési értéként `FALSE`-ot (sikeres betöltés esetén ezzel szemben `TRUE`-t). Ebből következik, hogy interaktív munkamenetnél – magyarul, ha kézzel pötyögünk, meg hajtunk végre kódsorokat – a `library` tökéletes, viszont ha csomag-betöltést egy futó program maga végzi, akkor a `require` jobb lehet, mert gépi úton tudjuk kezelni a helyzetet, hogy volt-e az adott gépen telepítve a csomag, sikerült-e betölteni, és ennek fényében továbbmenni. De vigyázat: ha ezt a kezelést nem hajtjuk végre, akkor a `require` veszélyes lehet, mert azt okozhatja, hogy a futás nem áll meg azonnal, és csak később – praktikusán amikor az adott könyvtárból akarunk egy függvényt hívni – lesz baj, amit viszont így nehezebb lesz beazonosítani, hogy mi okoz. Mivel azonban egy csomag elérhetőségének az ellenőrzésére van más megoldás, a `requireNamespace` függvény, így van, aki azt javasolja, hogy `require`-t egyáltalán ne használjunk.

The following object is masked from 'package:grDevices':

```
cm
```

Ha ezt megtettük, akkor onnantól nincs szükség a `::` operátorra, a csomag valamennyi függvénye automatikusan, minden előtag nélkül elérhetővé válik! Azaz innentől működik ez a megoldás is:

```
rinvexp(10)
```

```
[1] 1.0082068 0.5516095 1.9032739 0.9029940 0.3909869 5.9075980 0.6155178  
[8] 2.0401943 1.4307625 3.5926271
```

Egy apró megjegyzés a teljes precizitás kedvéért. A négyespont operátor csak azokat a függvényeket teszi elérhetővé, amik ún. exportált függvényei a csomagnak. Az exportálás lényegében azt jelenti, hogy elérhetővé akarjuk-e tenni az adott csomagbeli függvényt. Ez elsőre meglepő lehet (miért raknánk egy függvényt egy csomagba, ha nem akarjuk elérhetővé tenni?!), de elképzelhetőek helyzetek, ahol ennek van értelme, például, mert csak egy belső célokat szolgáló segédfüggvényről van szó – az ilyeneknek lehet szerepe, a felhasználó számára is, mert *más* csomagbeli függvények meghívhatják, csak közvetlenül nem lehet őket kívülről meghívni. Ha valami oknál fogva mégis ezt szeretnénk, akkor a `:::` operátort kell használni, ezzel nem exportált függvény is hívható.

A nyitva maradt kérdés, hogy melyik a jobb megoldás, a `::` használata vagy a `library`-vel történő betöltés? A válaszhoz egy dolgot kell még tudni, azt, hogy mi történik akkor, ha a csomagban van egy olyan nevű függvény, ami már létezik. Hiszen minden további nélkül lehet egy `rnorm` nevű függvény definiálva egy csomagban! (Vagy akár két különböző csomagban.) A helyzet az, hogy ilyenkor, ha betöltjük a csomagot `library`-vel, akkor felülíródik a csomag függvényével a korábban már létező, ugyanolyan nevű függvény. Ha betöltünk később egy másik csomagot is `library`-vel, amiben van ugyanolyan nevű függvény, akkor meg azzal íródik felül. Az `rnorm` kicsit extrém példa, de mondjuk `filter` nevű függvény tényleg tömegével van különféle csomagokban. A dolog tehát egy olvashatósági problémát fog okozni: ha egyszerűen annyit látunk egy kódsorban, hogy `filter(x)`, akkor nem tudhatjuk, hogy ez *minek* a `filter`-je, melyik csomag `filter` nevű függvénye fog itt lefutni. A „beépített” `filter`? (Mert van ez is; igazából a `stats` csomagé.) A `dplyr` csomag `filter`-je? A `signal` csomag `filter`-je? A `seewave` csomag `filter`-je? A legrosszabb, hogy ezt nem is *lehet* megmondani pusztán ebből a kódsorból – mert azon fog múlni, hogy melyik csomagot töltöttük be utoljára! Tehát ezt csak úgy tudjuk eldönteni, ha visszamegyünk a kódban (akár több száz vagy ezer sort is adott esetben), megnézzük az *összes* `library` hívást, hogy definiál-e `filter`-t, és ez fogja eldönteni, hogy a kérdéses sorban pontosan mi is fut: a beépített `filter` vagy valamelyik csomagé? Ha több, `filter`-t definiáló csomagot is betöltünk `library`-vel, akkor még az is számítani fog, hogy milyen sorrendben töltjük be őket! Emiatt célszerűbb, ha ahol lehet, inkább a `::` operátort

használjuk. Ennek az egyetlen hátránya¹², hogy többet kell hozzá írni, és ha nagyon sokszor ismétlődik, az zavaró lehet, így végeredményben az ajánlás az, hogy `library`-vel *csak* azokat a csomagokat töltsük be, amik alapvetőek, amik függvényeit sokszor és intenzíven használjuk – a többi esetében jobb a `::` használata.

Egyetlen megjegyzés a végére: néha érdemes a csomagokat frissíteni! Hiszen a csomagok frissülnek a CRAN-en, ha a szerzőjük feltölt egy újabb változatot, de az nem kerül át automatikusan a gépünkre. Ezért érdemes rendszeresen megnyomni RStudio-ban a Tools / Update packages pontot, és az összes csomagot frissíteni. (Egyébként végeredményben ez is az `install.packages`-t fogja meghívni.)

3.4. Függvények specifikációja és meghívása

Függvény úgy hívható meg, hogy megadjuk a nevét, majd utána gömbölyű zárójelben az argumentumát, vagy argumentumait:

```
rnorm(10)
```

```
[1] -0.8466682  1.9266213  0.3685584 -0.9143191  0.7370300 -1.9601099  
[7] -1.4180686 -0.8723739  0.2819766  0.3974036
```

Elképzelhető, hogy egy függvénynek egy argumentuma sincs, de a zárójelet ekkor is ki kell írni (azonnal becsukva, értelemszerűen). Ez fontos, ugyanis zárójel nélkül beírva a függvény nevét az R kiírja a függvény törzsét:

```
rnorm
```

```
function (n, mean = 0, sd = 1)  
.Call(C_rnorm, n, mean, sd)  
<bytecode: 0x0000021851892200>  
<environment: namespace:stats>
```

¹²Elvileg egy nagyon minimális teljesítmény-veszteséget jelent a `::` (mivel a háttérben egy függvényhívást jelent igazából), de ennek a legtöbb gyakorlati esetben nem lesz érdemi jelentősége: én most tettem egy próbát, az `actuar::rinvexp(10)` medián futásideje a gépemen 1,9 μ s, az `actuar` csomag `library`-vel betöltése után az `rinvexp(10)`-é 1,6 μ s. A különbség 300 ns, azaz ha egymillió alkalommal használjuk a `::` operátort egy kódban, az összesen 0,3 másodperc veszteséget fog jelenteni... Egy fokkal talán nyomósabb érv lehet a `library` mellett, hogy ha a szkriptünket azzal kezdjük, hogy az összes használt csomagot be-`library`-zzük, akkor egyrészt azonnal látszik, hogy mikre lesz szükség, másrészt, ha valamelyik nincs meg a futtató felhasználó gépén, akkor az rögtön az elején kiderül a hibából. A `library`-k használata nélkül ez csak az első `::`-nél bukna ki, ami nem elég: általános programtervezési elv, hogy ha valami baj van, akkor jobb, ha az minél előbb kiderül.

Függvényről sugó a kérdőjellel kapható: `?rnorm`. Ez a megoldás akkor használható, ha a függvény pontos nevét tudjuk, mert azt kell a kérdőjel után írni, ha nem tudjuk a pontos nevet, csak a név egy töredékét, akkor a két kérdőjel (`??rno`) használható; ez végigkeresi az összes sugó-oldalt a beírt töredék után, és listát ad róluk¹³.

Ami nekünk most különösen fontos a sugóban, az a függvény ún. specifikációja. (Ha saját magunk írtuk a függvényt, akkor ezt a specifikációt mi magunk kellett hogy megadjuk – ahogy az az előző pontban szerepelt is.) Az `rnorm` esetén ez a következőképp néz ki:

```
rnorm(n, mean = 0, sd = 1)
```

Az argumentumok az `args` függvénnyel is megtudhatók:

```
args(rnorm)
```

```
function (n, mean = 0, sd = 1)
NULL
```

Ebben a következő elemek láthatóak:

- A függvény neve, esetünkben az `rnorm`. Utána, ahogy már volt róla szó, gömbölyű zárójelben következnek az argumentumok.
- Szintén volt róla szó, hogy az argumentumok száma tetszőleges lehet, a nullát is beleértve; jelen esetben a függvénynek három argumentuma van.
- Minden argumentumnak van egy neve, kötelező is, hogy legyen. Ez esetben az első argumentum neve `n`, a másodiké `mean`, a harmadiké `sd`.
- Egy argumentumnak lehet, de nem kötelező, hogy legyen ún. alapértelmezett értéke; ha van, akkor egyenlőségjel után szerepel az argumentum neve után. Esetünkben az `n` nevű argumentumnak nincs alapértelmezett értéke, a `mean`-nek és az `sd`-nek van, az előbbinek 0, az utóbbinak 1.

Természetesen nagyon fontos, és a sugó többi részéből ez ki is derül, hogy egyáltalán mit csinál a függvény, mire jó, mi a tartalma az egyes argumentumoknak, mi a visszatérési értéke, de mi most fókuszálunk a szintaktikára.

Kezdjük ott, hogy ha egy argumentumnak van alapértelmezett értéke, akkor azt az argumentumot nem kötelező megadni a hívás során, viszont aminek nincs, azt kötelező. Ezért van az, hogy az `rnorm(10)` lefut, noha csak egy argumentumot adtunk meg (mert a másik kettőnek van alapértelmezett értéke), viszont ha az `n`-et nem adjuk meg, akkor hibát ad a függvényhívás:

¹³Ez a helyi gépen keres, ebből fakadóan természetesen csak azokat a csomagokat tudja végigkeresni, amik telepítve vannak helyileg. Létezik három kérdőjeles változat (`???rno`) is, ami az R központi weboldalán keres, így – többek között – valamennyi csomagot végignézi, függetlenül attól, hogy helyileg telepítve van-e. Ez azonban egy külön csomag, az `sos` telepítését igényli (a három kérdőjel ugyanis igazából nem más, mint egy hozzárendelt rövidítés, ún. alias az `sos::findFn` függvényhez, ami ilyen keresést hajt végre).


```
rnorm()
```

Error in rnorm(): argument "n" is missing, with no default

Az alapértelmezett értékkel rendelkező argumentumokat tehát nem kötelező megadni, de természetesen lehet:

```
rnorm(n = 10, mean = 70, sd = 15)
```

```
[1] 58.04227 68.30336 68.76622 76.16074 77.80667 66.46833 76.45759 78.79206  
[9] 80.28687 36.22849
```

Mint látható, argumentumot úgy adunk meg a hívás során, hogy beírjuk a nevét, egyenlőségjelet teszünk¹⁴, majd utána leírjuk az értékét. Ami fontos, hogy a nevek elhagyhatóak, ez esetben az R abban a sorrendben rendeli hozzá a beírt értékeket az argumentumokhoz, amilyen sorrendben a specifikációban szerepelnek¹⁶. Vagyis a fenti hívás egyenértékű ezzel:

```
rnorm(10, 70, 15)
```

```
[1] 100.32062 76.10365 90.10084 85.51383 76.14612 68.26165 104.10674  
[8] 82.26022 89.90167 54.06430
```

Ezt voltaképp már korábban is láttuk: az `rnorm(10)` ezért működött minden név megadása nélkül is.

A nevek megadása lehetővé teszi, hogy más sorrendben soroljunk fel argumentumokat, mint a specifikációban szerepelnek:

¹⁴Az értékadásnál említettem, hogy a nyíl helyett szinte felcserélhetően használhatnánk egyenlőségjelet is, és hozzátettem, hogy az R-es szokás az, hogy az egyenlőségjelet másra használjuk. Akkor most már elárulhatom: erre! A szokás az, hogy az értékadásnál nyilat írunk, függvény argumentumának megadásakor egyenlőségjelet. A kettő között a különbség minimális¹⁵.

¹⁶Hogy teljesen precíz legyek, az algoritmus a következő: először fogja az elnevezett argumentumokat, és megkeresi, hogy *pontosan* olyan névvel van-e paraméter, ha igen, akkor megfelelteti őket egymásnak. Ha marad még hívásban szereplő argumentum, aminek nem feleltetett meg paramétert, akkor tovább megy: ezt követően – és ez meglepő lehet – részleges egyezéseket is keres az elnevezett argumentumok között: megnézi, hogy van-e olyan, ahol a hívásban szereplő név egyezik a paraméter nevének az *elejével*. Például ha a hívásban `at1` néven hivatkozunk egy argumentumra az át fog adódni az `atlag` paraméternek (feltéve, hogy nem adtunk át `atlag` nevű argumentumot is, mert az az első lépésben már „megenné” az `atlag` nevű paramétert). Végezetül, ha még így is van nem összerendelt argumentum és paraméter, akkor az összes megmaradt argumentumot sorrendben osztja ki a megmaradt paramétereknek.

```
rnorm(sd = 15, n = 10, mean = 70)
```

```
[1] 54.47091 67.20102 64.82426 82.74180 80.33191 81.93191 56.88048 75.92414  
[9] 96.14272 55.22394
```

Vagy, hogy átugorjunk egy argumentumot:

```
rnorm(10, sd = 15)
```

```
[1] -10.308333 -19.897336 15.972754 -10.818097 -4.391853 2.777714  
[7] 2.948940 -12.195906 -16.289317 13.461429
```

Az R-es gyakorlat az, hogy az első néhány, mondjuk 3-4 argumentumtól eltekintve *akkor is* írjuk ki a neveket, ha egyébként sorrendben adtuk meg őket a hívás során. Ennek nem a kód futtathatóságához, hanem az olvashatóságához van köze: átlagos R-hez értő embertől elvárható, hogy – különösen a gyakran használt függvényeknél – az első néhány argumentumról tudja, hogy mi a jelentésük, de a továbbiakról már nem feltétlenül, így az olvasónak segítség, ha ezeknél feltüntetjük a nevet, még akkor is, ha az R-nek nem kellene, mert sorrendben jönnek. Ez azért van így, mert ezek általában beszélő nevek (ez az `rnorm` példáján is jól látszik!); ha majd mi magunk definiálunk függvényt, akkor is törekedjünk rá emiatt, hogy mi is ilyen beszélő neveket adjunk.

Egy utolsó dolgot kell még a fentiek kapcsán megbeszelnünk. Az eddigi leírásból úgy tűnhet, hogy a függvényeknek mindig adott, rögzített, előre ismert számú argumentumuk van – annyi, amennyit a specifikációban felsoroltunk. Az eddigi példákban ez valóban így volt, de gondoljunk bele, mi a helyzet mondjuk a `c`-vel? (Eddig nem mondtam, de természetesen ez is egy függvény!) Hogyan lehetséges, hogy működik a `c(1, 2)` és a `c(1, 2, 3)` is? A válasz az, hogy az R megenged egy speciális argumentumot, a három pontot¹⁷: a `c` függvény specifikációja úgy néz ki, hogy `c(...)`. A három pont azt jelenti: tetszőleges számú argumentum. Ilyen esetben tehát azt mondjuk, hogy mi magunk sem tudjuk, hogy hány argumentumot kapunk, minden teljesen azon múlik, hogy a felhasználó hogyan hívja meg a függvényt – ami szerepel a meghívásban, az fog átkerülni a `...` alatt, legyen az 0 argumentum, 1, vagy 100. A felhasználó megteheti, hogy a `...` helyén 0 argumentumot ír be a függvény meghívásakor, megteheti, hogy 1-et, megteheti, hogy 100-at. (Ha majd saját függvényt írunk: ilyenkor a függvényen belül szintén `...` névvel hivatkozhatunk arra, hogy mit kaptunk a hívás során. Annyit kell tudni erről, hogy ez egy elég speciális elem, első lépésben szinte mindig listává alakítjuk: `list(...)` már egy szokásos lista lesz, amit innentől a hagyományos módon, megszokott listaként használhatunk.) A `...`-ban átadott argumentumoknak lehet neve, de ez nem kötelező. A `...` vegyíthető a „szokásos”

¹⁷Angolul a neve ellipsis; semmi köze az ellipszishez, egyszerűen arról van szó, hogy angolul, mármint a nyelvészetben, így hívják a – tipikusan mondat végi – három pontot. Az R nemes egyszerűséggel „dot-dot-dot”-nak is hívja.

argumentumokkal: egy függvénynek nézhet úgy ki a specifikációja, hogy `f(x, ..., y = 1, z = 2)`. Ez azt jelenti, hogy az első egy kötelező argumentum, `x` névvel, utána jön tetszőleges, és előre nem ismert számú argumentum (akár 0 is): hogy itt mit kap a függvény, az teljesen azon múlik, hogy a felhasználó hogyan hívja meg. Ezután egy `y` nevű argumentum következik 1 alapértelmezett értékkel, végül egy `z` nevű 2 alapértelmezett értékkel. Ennek megfelelően az `f(1)` esetén az `x` értéke 1 lesz, a `list(...)` egy üres lista, `y` pedig 1, míg `z` értéke 2. Az `f(1, z = 10)` annyiban tér el ettől, hogy `z` értéke 10 lesz. Végezetül az `f(1, 2, a = 3, 4, z = 10)` hívásnál `x` értéke 1 lesz, a `list(...)` egy háromelemű lista lesz 2, 3 és 4 értékekkel (amiből a középsőnek `a` a neve, a másik kettőnek nincs neve), `y` értéke 1, `z` értéke 10. (Mint látható, ilyenkor, ha `y`-nak vagy `z`-nek szeretnénk beállítani az értékét, akkor kötelező megadni a nevét a hívásban, különben az R azt hinné, hogy a megadott argumentum a `...` része.)

Zárásként még egy apróság: bizonyos esetekben felmerül a kérdés, hogy mi a teendő akkor, ha úgy kell egy függvényt meghívunk, hogy mi magunk sem tudjuk előre (tehát a programkód írásakor) az argumentumait, például mert egy másik függvény állítja elő. Ilyenkor célszerű ezeket egy listába helyezni, az R ugyanis kínál egy megoldási lehetőséget erre:

```
fuggvenyargs <- list(n = 10, mean = 70, sd = 15)
do.call(rnorm, fuggvenyargs)
```

```
[1] 60.45549 57.95583 64.31685 68.38031 75.55367 56.76318 64.24478
[8] 106.54296 61.37858 63.88538
```

A `do.call` tehát meghívja az első argumentumban megadott függvényt a második argumentumban megadott argumentumokkal. A `do.call` lényegében elválasztja egymástól a függvény nevét és argumentumait (ami az `rnorm(sd = 15, n = 10, mean = 70)` típusú hívásnál össze van gubancolódva); ezzel megoldást adva a fentiekben említett helyzetre is. Ha a függvény argumentumai között `...` van, a `do.call` akkor is működik, vagyis az átadott lista hossza nem kötelező, hogy mindig ugyanaz legyen.

3.5. Az R-ben minden utasítás függvényhívás

Érdemes még egy dologról beszélni, ami látszólag egy erősen technikai aspektus, de valójában fontos következményei vannak: arról, hogy az R-ben minden utasítás igazából függvényhívás.

Ennek első ránézésre pár dolog ellentmond; nézzük most meg ezeket közelebbről. Kezdjük kapásból ezzel:

```
1 + 2
```

```
[1] 3
```

Ez hogy lenne már függvényhívás?! Nincs is benne függvénynév, pláne nincs gömbölyű zárójel, meg argumentumok...! – mondhatná valaki. Azonban a helyzet az, hogy de, van benne függvénynév, a `+`. Ez igenis egy függvény neve, nyugodtan leírhatjuk szokásos formában is, egyedül arra kell figyelni, hogy a `+` karakter nem felel meg a változónevekre vonatkozó, korábban látott szabályoknak – úgynevezett nem szintaktikus név – ezért a szokásos formájú hívásnál backtick-ek közé kell tenni:

```
`+`(1, 2)
```

```
[1] 3
```

Egyszerűen arról van szó, hogy az R ad nekünk annyi segítséget, hogy megengedi, hogy a zárójelet nem kell kitenni, de ami még fontosabb, hogy lehetővé teszi, hogy ilyen, úgynevezett operátorok a szokásos módon, az argumentumok között szerepeljenek. (Ezt hívják amúgy a programozáselméletben infix jelölésnek, míg a második, „szokásos függvényhívás” formát prefix jelölésnek.)

Mi a helyzet az értékadással? Mondjuk:

```
x <- 5  
x
```

```
[1] 5
```

Itt már talán nem okozok meglepetést: egyszerűen arról van szó, hogy a `<-` az igazából egy függvény. Úgyhogy igen, bármilyen meglepő, de a fenti igazából ezzel egyenértékű:

```
`<-`(x, 10)  
x
```

```
[1] 10
```

Az egyetlen, ami fontos, hogy ez egy mellékhatásos függvény, ahol a mellékhatás az, hogy a memóriában valami elhelyeződik.

A végére hagytam a talán legfontosabb példát. Mi a helyzet azzal, amit eddig úgy hívtunk, hogy „kiíratjuk egy változó értékét”? Például:

```
x
```

```
[1] 10
```

Na itt aztán végképp nincsen semmilyen függvényhívás...! – mondhatná valaki. De! Nagyon is van, a helyzet ugyanis az, hogy az R, miután végzett a kiértékelésekkel, az eredményre automatikusan meghívja a `print` függvényt. (Meglépő lehet, hogy kiértékelést említettem – mit kell abban kiértékelni, hogy `x`? – és ez valóban egyértelműbb lenne, ha azt írtam volna, hogy `exp(x)`, de valójában az `x` is kiértékelhető, egyszerűen azzal a szabállyal, hogy egy változó kiértékelte saját maga.) A fenti tehát igazából ezt jelenti:

```
print(x)
```

```
[1] 10
```

A `print` pedig egy mellékhatásos függvény, ahol a mellékhatás az, hogy az eredmény megjelenik a konzolon.

(Zárójel: tehát az `1 + 2` az nem csak ``+(1, 2)` valójában, hanem `print(`+(1, 2))`...!)

A dolog azért fontos, mert az R-ben van egy viselkedés, ami csak a fentiek ismeretében érthető meg. Töltsük be a példa-adatbázisunkat, és hajtsunk végre egy egyszerű statisztikai próbát:

```
data(birthwt, package = "MASS")
t.test(bwt ~ smoke, data = birthwt)
```

Welch Two Sample t-test

```
data:  bwt by smoke
t = 2.7299, df = 170.1, p-value = 0.007003
alternative hypothesis: true difference in means between group 0 and group 1 is not equal to
95 percent confidence interval:
 78.57486 488.97860
sample estimates:
mean in group 0 mean in group 1
 3055.696      2771.919
```

Hogy mi ez a próba, az most teljesen mindegy is (*t*-próbával vizsgáljuk, hogy a dohányzó és a nem-dohányzó anyukák újszülöttjeinek várható születési tömege szignifikánsan eltér-e), a fontos, hogy valamilyen komplexebb statisztikai elemzés. Mit látunk? Egy szép, ízlésesen elrendezett eredményt, benne a legfontosabb számokkal, sőt, még némi magyarázó szöveggel is kiegészítve. Ami most lényeges: úgy tűnik, hogy működik a szabály, hogy „ha nincs nyíl, akkor kiíratunk, de nem mentünk”: a kiíratás valóban megjelent, és új változó valóban ne jött létre. Ellenőrizzük le a fordított irányt:

```
teszteredmeny <- t.test(bwt ~ smoke, data = birthwt)
```

Továbbra is stimmel minden: most nem íratódott ki semmi, de létrejött az új változó. Ahogy vártuk! Most nézzük meg, mi van a változóban:

```
teszteredmeny
```

Welch Two Sample t-test

data: bwt by smoke

t = 2.7299, df = 170.1, p-value = 0.007003

alternative hypothesis: true difference in means between group 0 and group 1 is not equal to
95 percent confidence interval:

78.57486 488.97860

sample estimates:

mean in group 0 mean in group 1

3055.696 2771.919

Ez is pontosan megfelel a várakozásunknak: az van benne, amit korábban a kiíratásnál kaptunk. Mi más lenne? Nos, nézzük kicsit meg közelebbről ezt a változót:

```
str(teszteredmeny)
```

List of 10

\$ statistic : Named num 2.73

..- attr(*, "names")= chr "t"

\$ parameter : Named num 170

..- attr(*, "names")= chr "df"

\$ p.value : num 0.007

\$ conf.int : num [1:2] 78.6 489

..- attr(*, "conf.level")= num 0.95

\$ estimate : Named num [1:2] 3056 2772

..- attr(*, "names")= chr [1:2] "mean in group 0" "mean in group 1"

\$ null.value : Named num 0

..- attr(*, "names")= chr "difference in means between group 0 and group 1"

\$ stderr : num 104

\$ alternative: chr "two.sided"

\$ method : chr "Welch Two Sample t-test"

\$ data.name : chr "bwt by smoke"

- attr(*, "class")= chr "htest"

És akkor itt jön a meglepetés: ez a változó valójában egy 10 elemű lista...! De akkor hogyan lett a kiíratásakor ebből egy ilyen szöveg?! (A kiírt dolgok egy része, például ezek a magyarázó szövegek még csak benne sincsenek a változóban!) Hogy jelenhetett akkor ez meg a kiíratáskor?

A válasz a fenti output utolsó sorában rejlik. Amit itt látunk, hogy van egy `class` nevű attribútuma a változónak, ami `htest` értékre van beállítva. Mint tudjuk, bármilyen nevű attribútumot definiálhatunk egy változóhoz, de a `class` különlegesen viselkedik. A részletek megtárgyalása nélkül: valójában nem egyetlen `print` függvény van, hanem több tucatnyi, és az R-ben van egy belső mechanizmus, ami az alapján választja ki, hogy pontosan melyik fut le, hogy mi a kiíratott változó `class`-a. Ha `htest`, akkor igazából a `print.htest` fog lefutni¹⁸. Ez pedig így néz ki (figyeljük meg, hogy a `:::` operátort kell használnunk, mert ez a függvény ugyan benne van az – automatikusan betöltődő – `stats` csomagban, de nem exportált függvény, ami logikus is, mert kívülről direkte nem kell meghívunk):

```
stats:::print.htest
```

```
function (x, digits = getOption("digits"), prefix = "\t", ...)
{
  cat("\n")
  cat(strwrap(x$method, prefix = prefix), sep = "\n")
  cat("\n")
  cat("data:  ", x$data.name, "\n", sep = "")
  out <- character()
  if (!is.null(x$statistic))
    out <- c(out, paste(names(x$statistic), "=", format(x$statistic,
      digits = max(1L, digits - 2L))))
  if (!is.null(x$parameter))
    out <- c(out, paste(names(x$parameter), "=", format(x$parameter,
      digits = max(1L, digits - 2L))))
  if (!is.null(x$p.value)) {
    fp <- format.pval(x$p.value, digits = max(1L, digits -
      3L))
    out <- c(out, paste("p-value", if (startsWith(fp, "<")) fp else paste("=",
      fp)))
  }
  cat(strwrap(paste(out, collapse = ", "), sep = "\n")
  if (!is.null(x$alternative)) {
    cat("alternative hypothesis: ")
    if (!is.null(x$null.value)) {
      if (length(x$null.value) == 1L) {
```

¹⁸A `print`-nek természetesen van egy alapértelmezett variánsa is, a `print.default`, ez fut akkor, ha nincs `class`, vagy az adott `class`-hoz nincs külön definiált `print`.

```

        alt.char <- switch(x$alternative, two.sided = "not equal to",
            less = "less than", greater = "greater than")
        cat("true ", names(x$null.value), " is ", alt.char,
            " ", x$null.value, "\n", sep = "")
    }
    else {
        cat(x$alternative, "\nnull values:\n", sep = "")
        print(x$null.value, digits = digits, ...)
    }
}
else cat(x$alternative, "\n", sep = "")
}
if (!is.null(x$conf.int)) {
    cat(format(100 * attr(x$conf.int, "conf.level")), " percent confidence interval:\n",
        " ", paste(format(x$conf.int[1:2], digits = digits),
            collapse = " "), "\n", sep = "")
}
if (!is.null(x$estimate)) {
    cat("sample estimates:\n")
    print(x$estimate, digits = digits, ...)
}
cat("\n")
invisible(x)
}
<bytecode: 0x000002185191a740>
<environment: namespace:stats>

```

Nem fontos, hogy a fenti kód minden egyes részét pontosan értsük, a fontos az összkép: *ez* tartalmazza a szép, ízléses, magyarázott kiíratás sémáját! Ebben van benne az elrendezés, az összekötő szövegek stb., természetesen úgy, hogy a dolog hivatkozik a kiírandó változó megfelelő számaira – amik ugyebár a lista egyes elemei. Fix szöveggént kiírja, úgy sorba, hogy „sample estimates”, azaz a becslések a mintából, utána pedig jön az `x$estimate`, vagyis, hogy „vedd ki az `x`-ből – ami a kiírandó változó – az `estimate` nevű elemet”. És menjünk vissza, nézzük meg: valóban az lesz a lista `estimate` nevű elemében, ami megjelenik a „sample estimates” után! A (megfelelő) `print` tartalmazza a sémát, a konkrét számok pedig kitöltődnek a ki-`print`-elt változóból, mint listából.

Egyébként kipróbálhatjuk, hogy ha rákényszerítjük, hogy ne a `testreszabott`, hanem az alapértelmezett `print` függvényt használja az R a kiíratáshoz, akkor valóban úgy kapjuk meg a `teszteredmeny` változó tartalmát, mint egy szokásos lista:


```
print.default(teszteredmeny)
```

```
$statistic
      t
2.729886

$parameter
      df
170.1002

$p.value
[1] 0.007002548

$conf.int
[1]  78.57486 488.97860
attr("conf.level")
[1] 0.95

$estimate
mean in group 0 mean in group 1
      3055.696      2771.919

$null.value
difference in means between group 0 and group 1
      0

$stderr
[1] 103.9519

$alternative
[1] "two.sided"

$method
[1] "Welch Two Sample t-test"

$data.name
[1] "bwt by smoke"

attr("class")
[1] "htest"
```

Így már jobban érthető a listák jelentősége is, amit a korábbi fejezetben nem tudtam pontosan

elmondani: a listák rengetegszer előfordulnak – akkor is, amikor mi nem látjuk! Hiszen ebben a *t*-próbás példában sehol nem bukkantak fel explicite listák, nem kellett listát kezelünk, azt sem kellett tudnunk, hogy mi az, hogy lista – de a háttérben nagyon is volt szerepük, listák mozogtak a különböző függvények között. A kicsit is komplexebb statisztikai számítások nagyon gyakran listát adnak vissza (még ha ezt mi nem is látjuk), ami teljesen logikus ha meggondoljuk, hiszen tipikus, hogy ezek különböző típusú adatokat tartalmaznak. Ez így van a *t*-próbás példában is, ahol előfordul szám, szöveg, egyelemű és többelemű vektor. Az ilyenek tárolására pont a lista a megfelelő adatszerkezet.

A `class`-t természetesen mi magunk is kihasználhatjuk a saját függvényeinkben. A fentiek ugyanis azt jelentik, hogy ha beállítunk egy `class`-t a függvényünk által visszaadott listára és definiálunk egy `print`-et arra a `class`-ra, akkor a függvényünk eredménye rögtön szépen megformázva fog megjelenni – anélkül, hogy a felhasználónak bármit tennie kellett volna (vagy akár csak tudna erről az egésztől)!

4 Az R programozása

Az eddigieket felhasználva elkezdhetünk megismerkedni az R programozásának alapvető módszereivel és fortélyjaival. Ezek egyszerűbb problémák megoldására önmagában alkalmasak, a bonyolultabb feladatok esetén pedig építőelemként szolgálnak, melyek használata lehetővé teszi komplex programok létrehozását.

A következőkben mindenhol a már látott, születési tömeges adatbázist fogjuk példaként használni:

```
data(birthwt, package = "MASS")
birthwt$race <- factor(
  birthwt$race, levels = 1:3,
  labels = c("Kaukázusi", "Afro-amerikai", "Egyéb"))
```

4.1. Ismétlődő feladat kezelése saját függvénnyel

Kezdjünk egy egyszerű feladattal: határozzuk meg az átlagos születési tömeget a különböző rasszhoz tartozó anyák körében!

Kezdjük a kaukázusi rasszal. A feladat valóban nem túl bonyolult: elsőként le kell szűkíteni az adatbázist (ehhez használjuk a logikai vektorral történő indexelést, ahol a vektort természetesen gépi úton, összehasonlítással állítjuk elő), kiválasztani a megfelelő oszlopot, majd venni az átlagát:

```
mean(birthwt[birthwt$race == "Kaukázusi",]$bwt)
```

```
[1] 3102.719
```

A feladat teljesen hasonlóan oldható meg a másik két rasszal:

```
mean(birthwt[birthwt$race == "Afro-amerikai",]$bwt)
```

```
[1] 2719.692
```

```
mean(birthwt[birthwt$race == "Egyéb"],$bwt)
```

```
[1] 2805.284
```

E kódsorokra ránézve remélhetőleg mindenkiben megszólal a vészcsengő: ebben rengeteg redundancia, ismétlődés van. (Ha valakinek ez így nem lenne kellően drámai, akkor gondoljon arra, hogy mi van, ha nem 3, hanem 30 kategóriánk van!) De erre már láttuk a megoldást: saját függvényt kell írni! Jelen esetben ez így nézhet ki:

```
racemean <- function(race) {  
  mean(birthwt[birthwt$race == race,$bwt])  
}
```

Ha valaki még nem írt ilyet, akkor egy dolog szokott nehézséget jelenteni, az argumentum kezelése. Eleinte ugyanis furcsa lehet, hogy megjelent egy `race` változó (mi az értéke? mit jelent ez? sehol nem definiáltunk ilyet!). Ha valakit ez megzavarna, akkor egyetlen dologra kell emlékeznie: mindenhol, ahol `race`-t lát, oda kell képzelnie, hogy "`Kaukázusi`"! (Vagy bármelyik másik kategóriát.) A válasz a kérdésre ugyanis az, hogy ennek az az értéke, amit a felhasználó megad a függvény hívásakor (ami persze bármi lehet, ezért lesz ez változó); arról pedig az R gondoskodik, hogy valóban ez kerüljön ebbe a változóba. Igen, mi nem definiáltunk ilyen változót, de a függvény futásának idejére lesz ilyen, pontosan ugyanúgy, mint egy általunk létrehozott változó, ezt az R fogja intézni. Erre gondolva kell megírni a függvényt.

E saját függvény használatával így nézhet ki a probléma megoldása:

```
racemean("Kaukázusi")
```

```
[1] 3102.719
```

```
racemean("Afro-amerikai")
```

```
[1] 2719.692
```

```
racemean("Egyéb")
```

```
[1] 2805.284
```

4.2. Ciklusszervezés

A fenti megoldás nagyon sokat javított a redundancián, de azért egy kis hiányérzetünk maradhat: a `racemean` így is háromszor van leírva. Miért? Lényegében arról van szó, hogy ugyanazt a műveletet kell többször egymás után végrehajtani (csak más adatokon). Az ilyen ismétlődő – szó szerint azonos, vagy legalábbis hasonló – kódok végrehajtása megint csak egy redundancia-probléma. És a megoldás itt is ugyanaz: ne kézzel másolgassuk le többször egymás alá! Ha azonos műveletet kell végrehajtani, akkor csak adjuk meg, hogy mit és hányszor, ha pedig hasonlót, akkor legyen egy változó, ami különböző értékeket vesz fel, megtestesítve az eltérést a többszöri futtatások között, és egyszer adjuk meg a kódot (ami természetesen függeni fog ettől a változótól). Az ilyet szokták a programozásméletben ciklusnak vagy iterációnak hívni; nagyon sokszor merül ez fel a gyakorlatban. Az R egyik nagyon fontos eltérése más programnyelvektől, hogy hogyan kezeljük az ilyen helyzeteket.

Más programnyelveken általában explicite, kézzel kell megszerveznünk a ciklust: meg kell adnunk a kódrészletet amit újra meg újra megismétlünk, meg kell adni, hogy ez az ismételtetés hogyan történjen (milyen változó és hogyan változzon közben, meddig tartson a ciklus), az R-nél azonban nem ez a helyzet: az esetek túlnyomó többségében egy függvényt, az `apply`-függvénycsalád valamelyik tagját kell használni erre a célra. Itt például bevethetjük az `lapply`-t:

```
lapply(c("Kaukázusi", "Afro-amerikai", "Egyéb"), racemean)
```

```
[[1]]  
[1] 3102.719
```

```
[[2]]  
[1] 2719.692
```

```
[[3]]  
[1] 2805.284
```

Mi történik itt? Az `lapply` egy kétargumentumú függvény, az első argumentuma egy vektor, a második egy függvény. Az `lapply` azt csinálja, hogy a második argumentumban megadott függvényt egyesével ráereszti az első argumentumban megadott vektor minden egyes elemére (azaz meghívja a függvényt rájuk, mint argumentumra), és a kapott visszatérési értékeket összefűzi egy listába. (Innen az `l` betű a függvény nevének az elején: ez mindenképp listát ad vissza.) Ezzel a módszerrel lényegében létrehoztunk egy ciklust.

Gyakran használjuk még az `sapply`-t is, itt az `s` betű arra utal, hogy *simplified*, azaz egyszerűsített: ha az `lapply` olyan eredményt ad, ahol a lista valamilyen egyszerűbb objektumra konvertálható, akkor az `sapply` ezt megteszi. Itt erről van szó, hiszen a lista minden eleme

egyetlen szám, így az egyszerűbb objektum kézenfekvő lesz: egy vektor¹. Az `sapply`-jal valóban erre jutunk:

```
sapply(c("Kaukázusi", "Afro-amerikai", "Egyéb"), racemean)
```

Kaukázusi	Afro-amerikai	Egyéb
3102.719	2719.692	2805.284

És íme, a végleges, redundancia-mentes, elegáns, R-stílusú megoldás²!

Egy pillanatra visszatérnék még a „más programnyelvekben szokásos megoldások” kitételre. A legtipikusabb példa a `for`-ciklus; ez elvileg R-ben is megvalósítható: meg kell adni egy vektort, egy változónevet³ és egy kódot, az R pedig újra meg újra lefuttatja a kódot úgy, hogy a változó közben egyesével felveszi a vektorban foglalt értékeket⁴. A fenti példa ezért így nézhet ki `for`-ciklussal:

```
for(race in c("Kaukázusi", "Afro-amerikai", "Egyéb")) {  
  racemean(race)  
}
```

A szépséghiba, hogy így csak kiíratjuk az értékeket, de nem mentettük el változóba. Ezt megtehetjük például így, létrehozva egy – eleinte üres – számvektort az eredmények tárolására, és minden ciklusban hozzáfűzve az új eredményt a vektor végére:

```
res <- numeric()  
for(race in c("Kaukázusi", "Afro-amerikai", "Egyéb")) {  
  res <- c(res, racemean(race))  
}  
res
```

¹Ez természetesen nem mindig van így: lehet, hogy a listában mondjuk adatkeretek vannak, akár az is előfordulhat, hogy az egyik elem egy szám, a másik egy adatkeret. Ilyenkor az `sapply` nem egyszerűsít, hiszen nem is tud. Az sem biztos, hogy mindig vektorra egyszerűsít: ha a lista elemei vektorok, akkor mátrixszá fog egyszerűsíteni. Egyébként ez az oka annak, hogy van, aki nem ajánlja az `sapply` használatát nem interaktív kódban: nem tudható előre, hogy mi lesz az eredményének a típusa. Ezt a problémát `vapply` oldja meg, ahol ezt a típust meg kell adni, így a meghívása biztonságosabb – és kicsit gyorsabb is – mégis, ritkán használják.

²Amely megoldás ebben az esetben egyébként teljesen felesleges, mert erre a konkrét szituációra van egy másik, kész R-függvény – a `tapply` – de itt a cél csak a módszerek demonstrációja volt.

³Vigyázat, ha ilyen nevű változó már létezik, akkor az értéke felül fog íródni a `for`-ciklus végrehajtása során azokkal az értékekkel, amiket a változó a ciklusban felvesz, tehát, a vektorban lévő elemekkel.

⁴Látható tehát, hogy egy megszorítás van: előre és pontosan meg kell adni, hogy a ciklusváltozó milyen értékeket fog felvenni. A programozáselméletben ezt gyakran igazából `foreach`-ciklusnak hívják, és a `for`-ciklus nevet fenntartják arra az esetre, amikor csak azt adjuk meg, hogy mettől-meddig menjen a ciklusváltozó (esetleg kiegészítve azzal, hogy milyen lépésközzel), vagy megadjuk, hogy mi legyen a kezdőértéke, milyen műveletet hajtsunk végre vele minden iterációban, és mikor álljon meg a ciklus.

```
[1] 3102.719 2719.692 2805.284
```

Ez a megoldás működik, de nagyon szerencsétlen és kerülendő: a probléma az, hogy ez esetben a R-nek minden egyes ciklusban újra kell foglalnia a memóriát a `res` számára és átmásolnia az új helyre; ez veszteség, lassú lesz. Éppen ezért ilyen esetekben mindig az a jó megoldás, ha preallokálunk, tehát előre lefoglaljuk a memóriát a `res` végleges méretével, és a ciklusban mindig a megfelelő helyre tesszük az eredményt:

```
res <- numeric(length(races))
for(i in seq_along(races)) {
  res[i] <- racemean(races[i])
}
res
```

```
[1] 3102.719 2719.692 2805.284
```

(A `seq_along` funkciójában teljesen hasonló az `1:length(races)`-hez, ami valószínűleg az ember első gondolata lenne a fenti helyzetben. A `seq_along` azért jobb, mert ha a vektor hossza véletlenül nulla, akkor az utóbbi megoldás az `c(1, 0)` vektoron futtatná végig a ciklust, hiszen a `:` csökkenő sorozatot is tud generálni, míg a `seq_along` – helyesen – semmin, ugyanis a `seq_along(x)` hossza mindig azonos lesz `x` hosszával. A fenti esetben a dolognak nincs jelentősége, hiszen pontosan tudjuk mi lesz a `races`, de jobb a hibaállóbb megoldást megszokni.)

Talán már a fenti kód is sugallja, hogy miért jobb az `sapply` a maga 1, azaz egy sorával... A különbség oka egyszerűen az, hogy az `sapply` megcsinál helyettünk egy sor dolgot, amit a `for` ciklus esetén nekünk, kézzel kellett elintéznünk: memóriát foglal, végigmegy az elemeken, gondoskodik róla, hogy minden a megfelelő helyre kerüljön stb. Mi ennek az oka? Az `sapply` írói okosak, de legalábbis kedvesek voltak, hogy nekünk kevesebb munkák legyen, a `for` írói viszont megsavanyodott programozók, akik szeretnék, ha sokat dolgozhatunk? Nem. Ez azért van, mert a `for` flexibilisebb: igaz, hogy az `sapply` ezeket mind megteszi helyettünk, viszont cserében mást nem is választhatunk, a `for` esetén viszont megtehetjük, hogy minden második elemet mentjük csak el egy vektorba, ettől függetlenül minden harmadikat kiíratjuk, és minden negyediket dallamá konvertáljuk, majd feltöltjük a Youtube-ra. Vagy, hogy mondjak egy földhözragadtabb (és talán a gyakorlatban is sűrűbben előforduló...) példát: `for`-ciklussal tudunk hivatkozni a *többi* elemre, például az előző iterációra, `sapply`-jal ez nem tehető meg. A konklúzió tehát egyszerű: ha nekünk *pont* arra van szükségünk, amit az `sapply` kínál – tehát az egymástól függetlenül kiszámítható visszatérési értékek összefogva egy vektorba – és nagyon sokszor erre lesz, mert nem véletlen, hogy az `sapply` pont ezt kínálja, akkor használjuk azt, ha nem erre, hanem valami általánosabbra, akkor használjuk a `for`-t. Megjegyzem, hogy ez egyébként egy teljesen általános tervezési elv is a programozásban: mindig a lehető

legkevésbé flexibilist eszközt használjuk azok közül, amik már elég flexibilisek a problémánk megoldásához.

Valójában egyébként nem egyszerűen a rövidség a fő kérdés⁵, hanem az olvashatóság, értelmezhetőség, de ami valójában még ennél is fontosabb, hogy az **apply** család sokkal jobban megfelel az R filozófiájának, mint a **for**-ciklus. Erről a következő pontban lesz részletesen szó.

A dolog végkonklúziója egyszerűen összefoglalható: R-ben ne használjunk **for**-ciklust! Szinte minden esetben igaz, hogy a **for**-ciklus kiváltható megfelelő **apply**-jal, és ki is váltandó.

Az egyetlen kivétel, ha mellékhatásos számítást végzünk, például fájlba kell mentenünk. Ez esetben az **apply**-család nem igazán esik kézre (hiszen nincs is visszatérési érték, de legalábbis nem amiatt végezzük a számítást), így inkább elfogadható a **for**-ciklus is.

A más programnyelvekben megszokott hátultesztelő ciklus nincs az R-ben, előltesztelő van⁶. A dolgot nem részletezem, hiszen már a **for**-ciklus használata is kerülendő, így erre végképp ritkán van szükség.

Egyetlen megjegyzés a végére. A fentiekben végig azt feltételeztem, hogy nem szó szerint ugyanazt a műveletet kell többször végrehajtani (igen, a kód ugyanaz volt, de az argumentum változott). Ha mégis erre volna szükség, akkor a **replicate** függvényt használhatjuk, mely a második argumentumban várja a többször megismétlendő kódot (egyetlen kifejezést, tehát ha nem egy soros a kód, akkor kapcsos zárójelekkel blokkot kell képezni), amit az első argumentumban megadott darabszámúszor lefuttat, és eredményeket összefűzi egy megfelelő objektumba (ha egyetlen számot adunk vissza, akkor vektorba). Ezt gyakran használjuk egyszerű Monte Carlo-szimulációkban. Például egy 30 elemű minta mintaátlagának viselkedését így szimulálhatjuk:

```
replicate(10, {  
  minta <- rnorm(30)  
  mean(minta)  
})
```

```
[1] -0.09195803 -0.61039553  0.08862312  0.01789136 -0.28837370 -0.16466751  
[7]  0.19898881  0.12339722  0.22290081  0.18875808
```

⁵Szokták e mellett még azt is mondani, hogy a gyorsaság szól az **apply** család mellett, mert a **for**-ciklus lassú, de ez már régen nem igaz: az R mostani verzióiban a **for**-ciklus már nagyjából ugyanolyan gyors.

⁶Két verzióban is. A **while** szokásos előltesztelő ciklus, a **repeat**-nél viszont nincs feltétel, ezért a végtelenségig futna, így értelme csak akkor van, ha manuálisan kiugrasztnak a ciklusból, erre a **break** nevű utasítás szolgál (ezt egyébként **for**-ciklusban is használhatjuk). A **for** ciklus mindig átírható **while**-ra, a **while** pedig mindig átírható **repeat**-re, de fordítva nem feltétlenül. Ezért elvileg lehet olyan helyzet, hogy nem tudunk **for**-t, vagy **while**-t használni, de ha tudunk, akkor használjuk inkább azokat – itt is előjön megint ugyanaz, hogy a legkevésbé flexibilis eszköz lesz a jobb választás.

4.3. Funkcionális programozás

Az előző pontban úgy vezettem be az **apply** családot, mint egy cseles programozástechnikai megoldást ciklusszervezésre. Valóban ez az egyik felfogás, de a mélyben sokkal fontosabb dolgok rejtőznek.

Ehhez kezdjünk egy tételmondatlal: bár az R – mint a legtöbb mai programozási nyelv – többféle programozási paradigmát támogat, ami legközelebb áll hozzá, az a funkcionális programozás. Hogy még egyértelműbben fogalmazzak, az R egy funkcionális programozási nyelv.

Mit jelent ez? A programozáselméletben szoktak beszélni két programozási alapfilozófiáról, programozási paradigmáról: az imperatív programozás az, ahol a programban konkrétan elő kell írunk, hogy mit csináljon a számítógép, a deklaratív programozási paradigma pedig az, ahol azt írjuk le a programban, hogy mi az elérendő cél, a megvalósítást ez alapján már a gép fogja kitalálni. A deklaratív paradigma egyik altípusa a funkcionális programozás. A nevét a függvényről (angolul function) kapta, és csakugyan, a függvényeknek e paradigmában kiemelt szerepük van. Itt muszáj rögtön egy pontosítást tennünk. Függvények más programozási paradigmában is vannak, természetesen, akár még kiemelt szerepük is lehet, még egy imperatív programban is, de a funkcionális programozási paradigmában ennél többről van szó: itt nem csak *léteznek* függvények, hanem *csak* függvények léteznek. E paradigma egyik sarokköve ugyanis, hogy *mindent* függvények hívásával oldunk meg. Több más jellemzője is van a funkcionális nyelveknek, lesz még néhányról itt is szó, de ez a legfontosabb. (Már itt is látszik, hogy miért mondtam, hogy ez az R valódi filozófiája – emlékezzünk csak vissza arra, hogy az R-ben minden függvényhívás...!)

Nézzünk egy egyszerű példát: ki akarjuk számolni az n pozitív egész szám faktoriálisát. Imperatív megoldás: legyen a **fact** változó értéke 1, majd fusson egy **for**-ciklus melyben az **i** ciklusváltozó megy 1-től n -ig egyesével, és a ciklus minden iterációjában cseréljük le **fact** értékét **fact * i**-re. Ez egy imperatív megoldás volt, pontosan előírtuk a számítógépnek, lépésről-lépésre, hogy mit kell csinálnia. Ugyanez funkcionális felfogásban: legyen **fact(n)** egy függvény azzal a definícióval, hogy 1 a visszatérési értéke ha $n = 0$, egyébként a visszatérési értéke $n * \text{fact}(n - 1)$. Az is szépen látszik, hogy ezt miért hívják deklaratív paradigmának (nem előírtuk, hogy mit kell csinálni a kiszámításhoz, csak deklaráltuk, hogy a faktoriálisnak milyen tulajdonsága van – és innentől a gépre bízunk, hogy ezt megoldja), de a mostani szempontunkból még érdekesebb, hogy mutatja, mitől funkcionális a paradigma: mindent függvények alkalmazásával oldunk meg, nincsen ciklus, még csak értékadás sincsen.

Fontos: a dolog célja egyáltalán nem feltétlenül az, hogy a kapott programunk gyorsabb legyen. Simán lehet, hogy végeredményben (gépi kód szintjén) a funkcionális megközelítéssel pont ugyanazt kapjuk, mint az imperatívval, sőt, elég tipikus, hogy a funkcionális még lassabb is. A cél az, hogy egyszerűbb, emiatt könnyebben megírható, kisebb valószínűséggel hibás, könnyebben ellenőrizhető, jobban optimalizálható kódunk legyen. Az előbbi példa is mutatja, hogy a funkcionális megközelítés átláthatóbb, egyszerűbben értelmezhető kódot eredményez,

amiről sokkal könnyebben és gyorsabban látszik, hogy mit csinál a program, vajon helyes-e az. (Képzeljük el mindezt egy faktoriális-számításnál százszor nehezebb problémánál!)

A funkcionális programozási paradigma kiemelten jól illeszkedik az adatelemzési, statisztikai feladatokhoz. Amint volt róla szó, az R mást is támogat, így nem kötelező R-ben ezt használni... de *értelmes*! (Valójában az, hogy más, nem tisztán funkcionális eszközöket is támogat, még akkor is jól jön, ha *alapvetően* funkcionális szemléletben kódolunk: a teljesen vegytisztán funkcionális megoldások lehetnek nagyon nyakatekertek, ezért a gyakorlatban sokszor megkönnyíti az életet egy csipetnyi nem-funkcionális elem hozzákeverése az egyébként funkcionális kódhoz.)

Egy második példa lehet épp az előző alfejezet, hiszen ott mivel váltottuk ki a `for`-ciklust? Függvény-alkalmazással! A `lapply`, `sapply` mind függvény volt, ezek alkalmazásával értük el pontosan ugyanazt a célt.

Itt érdemes megjegyezni azt is, hogy ezek egész pontosan milyen függvények voltak: olyanok, amik paraméterként egy másik függvényt vártak. Az ilyeneket szokták magasabbrendű függvénynek nevezni, és a dolog nem véletlen: ezek nagyon gyakran fordulnak elő a funkcionális paradigmában. Magasabbrendű függvénynek nevezzük azokat a függvényeket is, amik függvényt adnak visszatérési értéként, sőt, olyan is lehet, hogy a függvény argumentumként is függvényt vár, és visszatérési értéként is függvényt ad.

Itt érünk el a funkcionális paradigma egyik tipikus jellemzőjéhez: ebben a megközelítésben a függvények, úgy szokták mondani, „elsőrendű állampolgárok”. Függvény ugyanúgy változó, mint bármilyen más változó, függvény átadható argumentumként, azaz van függvény, ami bemenetként függvényt vár, sőt, írhatunk függvényt, ami függvényt ad vissza visszatérési értéként. Egyszóval: ahol lehet valamilyen változó, ott lehet függvény is.

Egy másik fontos jellemző a tiszta függvények alkalmazására törekvés. Idézzük fel a tiszta függvény definícióját: ez olyan függvény, aminek nincs mellékhatása és a visszatérési értékét determinisztikusan meghatározzák az argumentumai, vagyis ugyanahhoz az argumentumhoz mindig ugyanaz a visszatérési érték tartozik. Ha a kettő egyszerre teljesül, akkor nevezzük a függvényt tisztának. Az első feltétel megsértése miatt nem tiszta függvény például a `print` vagy a `write.csv2`, de az értékadás (`<-`) sem, a második feltétel megsértése miatt nem tiszta függvény az `rnorm`, de a `Sys.time` sem. (Itt utalnék vissza a korábbi megjegyzéseimre arról, hogy a gyakorlatban általában nem vagyunk vegytisztán funkcionálisak: képzeljünk el olyan adatelemzést, amiben nem írhatunk ki eredményt, nem adhatunk értéket változónak, vagy amiben nem generálhatunk véletlenszámokat...!) Nem lehet tehát mindig tiszta függvényeket használni, és ez nem is baj – ki kell használni annak az erejét, hogy az ilyen helyeken nyakatekertség helyett egy kicsi nem-funkcionális megoldást vetünk be – de a tiszta függvények elegánsabbak és jobban elemezhetőek.

5 Data table: egy továbbfejlesztett adatkeret

Amint volt már róla szó korábban, az adatkeret (data frame) az alapvető struktúra a feldolgozandó adatok, adatbázisok tárolására és kezelésére az R-ben. Noha ennek a célnak megfelel, számos téren kiegészíthető, továbbfejleszthető. Az évek alatt két nagy lehetőség kristályosodott ki és ment át széleskörű használatba, mely ilyen továbbfejlesztést jelent: a `dplyr` csomag és a `data.table` csomag. Jelen fejezet a `data.table` működését és jellemzőit fogja bemutatni, különös tekintettel a hagyományos data frame-mel való összevetésre.

A `data.table` csomag első verziója 2008-ban jelent meg, eredeti megalkotója Matt Dowle. Nagyon erőteljes, gyors, erősen optimalizált, némi gyakorlás után logikus, kompakt, konzisztens, könnyen lekódolható és jól olvasható szintaktikájú, jól támogatott¹ csomag. A `data.table`-nek semmilyen függősége nincs az R-en kívül (ott is törekednek a nagyon régi változatok támogatására is), így kifejezetten problémamentesen beépíthető R kódokba, csomagokba.

Központi weboldala: <https://rdatatable.gitlab.io/data.table/>. Github-oldala: <https://github.com/Rdatatable/data.table>. CRAN-oldala: <https://cran.r-project.org/web/packages/data.table/index.html>.

A `data.table` mint csomag egy azonos nevű új adatstruktúrát definiál; ez lényegében egy „továbbfejlesztett data frame”. Ez az új adatstruktúra, a data table egyrészt olyan lehetőségeket biztosít, amik valamilyen módon megvalósíthatóak lennének szokásos data frame-mel is, de csak lassabban/nehézkesebben/több hibalehetőséggel, másrészt elérhetővé tesz olyan funkciókat is, amik data frame-mel egyáltalán nem megoldhatóak.

A következőkben át fogjuk tekinteni ezek legfontosabb példait.

A gyakorlati szemléltetésekhez töltsük be a könyvtárat (a `data.table` nem jön az alap R installációval, így ha korábban nem tettük meg, elsőként telepíteni kell):

```
library(data.table)
```

Ebben a fejezetben a magyar Nemzeti Rákregiszter adatait² fogjuk példa adatbázisnak használni.

Elsőként töltsük be a következő fájlt, ami eleve `data.table` formátumban tartalmazza az adatokat:

¹<https://github.com/Rdatatable/data.table?tab=readme-ov-file#community>

²<https://github.com/ferenci-tamas/RakregiszterVizualizator>

```
if(!file.exists("RawDataLongWPop.rds"))
  download.file(paste0(
    "https://github.com/ferenci-tamas/RakregiszterVizualizator/",
    "raw/refs/heads/master/RawDataLongWPop.rds"),
    "RawDataLongWPop.rds")

RawData <- readRDS("RawDataLongWPop.rds")
```

Néhány esetben össze fogjuk vetni a `data.frame`-et a `data.table`-el, ehhez „minősítsük vissza” az adatbázist `data.frame`-mé, és ezt mentjük el egy új változóba:

```
RawDataDF <- data.frame(RawData)
```

Érdeemes ránézni egy `data.table` felépítésére:

```
str(RawData)
```

```
Classes 'data.table' and 'data.frame': 1313280 obs. of 7 variables:
 $ County      : chr  "Baranya megye" "Baranya megye" "Baranya megye" "Baranya megye" ...
 $ Sex         : chr  "Férfi" "Férfi" "Férfi" "Férfi" ...
 $ Age         : num  0 0 0 0 0 0 0 0 0 0 ...
 $ Year        : num  2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 ...
 $ ICDCode     : chr  "C00" "C01" "C02" "C03" ...
 $ N           : int  0 0 0 0 0 0 0 0 0 0 ...
 $ Population: num  9876 9876 9876 9876 9876 ...
 - attr(*, ".internal.selfref")=<externalptr>
 - attr(*, "sorted")= chr [1:4] "County" "Sex" "Age" "Year"
```

Ami feltűnhet, hogy az objektumnak egyaránt van `data.table` és `data.frame` osztálya, egyebekben azonban a fenti információk megfelelnek egy `data.frame` által mutatott felépítésnek. A két osztály jelenléte egyfajta visszafele kompatibilitást³ jelent: egy `data.table` olyan számítógépen is betölthető, ahol nincs `data.table` csomag, és működni fog (természetesen csak mint hagyományos `data.frame`). Ezen túl az is igaz ennek következtében, hogy olyan függvénynek, ami `data.frame`-et vár mindig átadható `data.table` is.

³Arra azért vigyázni kell, hogy van példa arra, hogy pontosan ugyanaz a hívás mást ad vissza a `data.frame`-nél és `data.table`-nél. Egyébként ez a válasz arra a gyakran felmerülő kérdésre, hogy ha olyan jó a `data.table`, akkor miért nem győzik meg egyszerűen a fejlesztői az R fejlesztőit, hogy építsék be a tulajdonságait az R-es alap `data.frame`-be is. Egyébként volt példa ilyenre is, de az előbbi ok miatt ez nem lehet általános, hiszen ez azt jelentené, hogy meglevő kódok működése is megváltozna, ami végeláthatlan sok R kód működését ronthatná el. Ilyen módosítást ma már nem igen lehet megtenni a `data.frame`-mel.

Visszatérve a tábla felépítésére, a fentiek alapján már elmondható, hogy a tábla mit tartalmaz: az új rákos esetek előfordulását Magyarországon évenként (2000-től 2018-ig), megyénként, nemeként, életkoronként (ez 5 éves felbontású, tehát a 40 igazából azt jelenti, hogy „40-45 év”), és a rák típusa szerint. Ez utóbbi ún. BNO-kóddal⁴ van megadva: a Betegségek Nemzetközi Osztályozása (BNO, angol rövidítéssel ICD) egy nemzetközileg egységes rendszer, mely minden betegséghez egy kódot rendel. A kód első karaktere egy betű, ez a főcsoport; a rákos betegségek a C főcsoportban, illetve a D elején vannak, a második és harmadik karakter egy szám, ami konkrét betegséget vagy betegségcsoport azonosít; például C00 az ajak rosszindulatú daganata, C01 a nyelvgyök rosszindulatú daganata és így tovább⁵. Az esetek számát az N nevű változó tartalmazza, a háttérpopuláció lélekszámát⁶, tehát, hogy hány fő volt adott évben, adott megyében, adott nemben, adott életkorban – pedig a **Population**. (Ez tehát azonos lesz azokra a sorokra, amelyek csak a rák típusában térnek el, hiszen ezekre a háttérpopuláció lélekszáma természetesen ugyanaz.)

5.1. Sebesség és nagyméretű adatbázisok kezelése

Ez a probléma a legtöbb szokásos elemzési feladatnál nem jelentkezik, itt sem fogunk rá részletes példát nézni, de röviden érdemes arról megemlékezni, hogy a hagyományos adatkeret (data frame) adatstuktúra nem szerencsés, ha nagyméretű adatbázisokat kell kezelünk.

Az első probléma kapásból az adatok beolvasásánál fog jelentkezni: a `read.csv` (és társai) egész egyszerűen lassúak. Pár százezer sorig ennek semmilyen érzékelhető hatása nincsen, mert még így is elég gyors a beolvasás, így a legtöbb feladatnál ez a probléma nem jelentkezik, de millió soros, több millió soros adatbázisoknál, ha a tábla mérete több gigabájt vagy több tíz gigabájt, akkor a beolvasás a méret növekedtével gyorsan lassul, míg végül teljesen reménytelenné válik. A `data.table` definiálja az `fread` függvényt mely ezzel szemben villámgyors, és még ilyen méretű adatok beolvasásánál is elfogadható sebességet produkál. (Az `fread`-nek ezen kívül van pár további előnye is az R beépített beolvasó függvényeihez képest, olyan, amik kis méretű adatbázisoknál is érdekesek lehetnek, például nagyon okosan detektálja az oszlopelválasztókat és az oszloptípusokat.) Hasonló a helyzet kiírásnál: a `write.csv` és társai nagyon nagy adatbázisoknál elfogadhatatlanul lassúak lesznek, de a `data.table` könyvtár `fwrite` függvénye ilyenkor is jól működik.

A második probléma, hogy még ha valahogy be is olvastuk az adatbázist a memóriába, akkor is bajban leszünk az adattranszformációkkal: a data frame nincs túl jól optimalizálva ilyen szempontból, egy sor művelet nagyon lassú. Ismét csak: ennek kis, közepes és a legtöbb terület mércéje szerinti nagy adatbázisoknál nincs jelentősége, mert még így is gyors, de a nagyon nagy

⁴<https://icd.who.int/browse10/2019/en>

⁵A kód folytatható, ami finomabb felbontást ad, például a C00.0 a felső ajak külső felszínének daganata, a C00.1 az alsó ajak külső felszínének daganata stb., de a táblázatunk a háromjegyű besorolást tartalmazza.

⁶Ez ún. évközepi lélekszám, tehát az év alatti – folyamatosan változó – lélekszámok átlaga. Ezért lehet az értéke törtszám is.

adatbázisoknál bajban leszünk data frame-et használva. A data table ezzel szemben nagyon jól optimalizált, képest többmagú processzoroknál bizonyos műveletek párhuzamos végrehajtására is, így az adattorzformációs műveleteknél⁷ (aggregáció, táblaegyesítések, de akár új változó létrehozása) sokkal jobb sebességet tud produkálni.

A fentieket többféle benchmark vizsgálat⁸ is megerősíti.

5.2. Jobb kiíratás

A data frame kiíratásánál (tehát ha egyszerűen beírjuk, hogy `RawDataDF`, ami ekvivalens a `print(RawDataDF)` függvény meghívásával) az alapbeállítás az, hogy kiírja a konzolra az első jó sok sorát az adatbázisnak⁹. Ez nem túl praktikus: az 587. sor ismerete jellemzően nem sokat ad hozzá az első 586-hoz, cserében hosszasan kell görgetnünk a rengeteg sor miatt, hogy elérjünk a kiíratás tetejére, aminek viszont volna jelentősége, mert ott látjuk az oszlopok neveit. (Nem véletlenül gyakori, hogy sokan eleve a `head(RawDataDF)` típusú kéréssel íratják ki a data frame-eket!)

A data table alapértelmezett kiíratása okosabb, mert csak az első néhány és az utolsó néhány sort¹⁰ írja ki:

`RawData`

Key: <County, Sex, Age, Year>

	County	Sex	Age	Year	ICDCode	N	Population
	<char>	<char>	<num>	<num>	<char>	<int>	<num>
1:	Baranya megye	Férfi	0	2000	C00	0	9876.0
2:	Baranya megye	Férfi	0	2000	C01	0	9876.0
3:	Baranya megye	Férfi	0	2000	C02	0	9876.0
4:	Baranya megye	Férfi	0	2000	C03	0	9876.0
5:	Baranya megye	Férfi	0	2000	C04	0	9876.0

1313276:	Zala megye	Nő	85	2018	D06	0	4483.5
1313277:	Zala megye	Nő	85	2018	D07	0	4483.5

⁷Ezek egy részénél nem kell külön függvényt hívni, csak „maga a data table” gyorsabb lesz mint a data frame. Más részénél szükség van egy külön függvényre, például a táblaegyesítésnél a `merge`-re. De ez is gyorsabb lesz, aminek a hátterében az van, hogy a `data.table`-nek van saját, ugyanilyen nevű függvénye (`data.table::merge`), és ez fog a data frame-hez tartozó alapváltozat, tehát a `base::merge` helyett futni.

⁸<https://duckdblabs.github.io/db-benchmark/>

⁹Egész pontosan annyit, amennyi a `max.print` opció értéke; ez a `getOption("max.print")` paranccsal kérdezhető le. Az alapbeállítása tipikusan 1000.

¹⁰A precizitás kedvéért: ezt csak akkor teszi, ha a sorok száma nagyobb mint a `datatable.print.nrows` opció értéke, ami alapbeállítás szerint 100. De ez is logikus: kis adatbázisnál érdemes az egészet kiíratni, hiszen úgy is áttekinthető, nagyoknál lesz fontos csak az első néhány és az utolsó néhány sor kiíratása.

1313278:	Zala megye	Nő	85	2018	D09	0	4483.5
1313279:	Zala megye	Nő	85	2018	D30	0	4483.5
1313280:	Zala megye	Nő	85	2018	D33	0	4483.5

Természetesen láthatóak az oszlopfejlécek (változónevek) is, sőt, itt van még egy további apró fejlesztés: a data table kiírja az egyes oszlopok adattípusát is, standard rövidítéssel.

5.3. Kényelmesebb sorindexelés (sor-szűrés és -rendezés)

Data frame indexeléséhez szögletes zárójelet kell írunk a változó neve után, abba vesszőt tennünk, majd a vessző elé kerül az sor indexelése. Ezt tipikusan szűréshez használjuk. Például, ha ki akarjuk választani csak a 2010-es év adatait:

```
head(RawDataDF[RawDataDF$Year == 2010,])
```

	County	Sex	Age	Year	ICDCode	N	Population
961	Baranya megye	Férfi	0	2010	C00	0	9430
962	Baranya megye	Férfi	0	2010	C01	0	9430
963	Baranya megye	Férfi	0	2010	C02	0	9430
964	Baranya megye	Férfi	0	2010	C03	0	9430
965	Baranya megye	Férfi	0	2010	C04	0	9430
966	Baranya megye	Férfi	0	2010	C05	0	9430

Ez lényegében a „logikai vektorral indexelés” esete: a `RawDataDF$Year == 2010` egy adatbázissal sorainak számával azonos hosszúságú logikai vektor lesz.

Ha ki akarjuk választani 2010 évben a 40 évnél idősebbek adatait, akkor a logikai ÉS operátort (&) kell használnunk; ez egyúttal azt is szemlélteti, hogy a feltételek természetesen nem csak egyenlőségek lehetnek:

```
head(RawDataDF[RawDataDF$Year == 2010 & RawDataDF$Age >= 40,])
```

	County	Sex	Age	Year	ICDCode	N	Population
15553	Baranya megye	Férfi	40	2010	C00	0	13076
15554	Baranya megye	Férfi	40	2010	C01	0	13076
15555	Baranya megye	Férfi	40	2010	C02	0	13076
15556	Baranya megye	Férfi	40	2010	C03	0	13076
15557	Baranya megye	Férfi	40	2010	C04	0	13076
15558	Baranya megye	Férfi	40	2010	C05	0	13076

A dolog hasonlóan folytatódik, ha további feltételek vannak. Például 2010 évben a 40 évnél idősebb budapesti vagy Pest megyei férfiak körében előforduló vastagbélrákos (BNO-kód: C18) esetek kiválasztása:

```
head(RawDataDF[RawDataDF$Year == 2010 & RawDataDF$Age >= 40 &
  RawDataDF$County %in% c("Budapest",
    "Pest megye") &
  RawDataDF$Sex == "Férfi" &
  RawDataDF$ICDCode == "C18",])
```

	County	Sex	Age	Year	ICDCode	N	Population
146899	Budapest	Férfi	40	2010	C18	3	57445.5
148723	Budapest	Férfi	45	2010	C18	10	42410.0
150547	Budapest	Férfi	50	2010	C18	17	45329.0
152371	Budapest	Férfi	55	2010	C18	44	55633.5
154195	Budapest	Férfi	60	2010	C18	59	45170.0
156019	Budapest	Férfi	65	2010	C18	120	39588.0

A dolog tökéletesen működik, ámde nem túl kényelmes: folyton be kell írni a `RawDataDF$`-t a feltételek közé. A kód hosszú, lassabb megírni, és az olvashatóság is romlik. Fontos hangsúlyozni, hogy ez nem hagyható el, és teljesen igaza is van az R-nek, hogy nem hagyható el: `Year` nevű változó nem létezik, tehát teljes joggal ad hibát, ha előle – vagy bármelyik másik elől – elhagyjuk a data frame nevét.

Mégis: a gyakorlatban az esetek 99,99%-ában, ha egy változó nevére hivatkozunk miközben egy adatkeret sorindexelését végezzük, akkor azt természetesen úgy értjük, hogy *annak az adatkeretnek* az adott nevű oszlopa (és nem egy külső változó). Éppen emiatt a data table megengedi ezt a szintaktikát: ha pusztán egy változó nevére hivatkozunk, akkor ő megnézi, hogy nincs-e olyan nevű oszlopa az indexelt adattáblának, és ha van, akkor úgy veszi, hogy arra szerettünk volna hivatkozni. Éppen ezért az alábbi kód `data.frame`-mel nem, de `data.table`-lel működik:

```
RawData[Year == 2010 & Age >= 40 &
  County %in% c("Budapest", "Pest megye") &
  Sex == "Férfi" & ICDCode == "C18",]
```

Key: <County, Sex, Age, Year>

	County	Sex	Age	Year	ICDCode	N	Population
	<char>	<char>	<num>	<num>	<char>	<int>	<num>
1:	Budapest	Férfi	40	2010	C18	3	57445.5
2:	Budapest	Férfi	45	2010	C18	10	42410.0
3:	Budapest	Férfi	50	2010	C18	17	45329.0

4:	Budapest	Férfi	55	2010	C18	44	55633.5
5:	Budapest	Férfi	60	2010	C18	59	45170.0
6:	Budapest	Férfi	65	2010	C18	120	39588.0
7:	Budapest	Férfi	70	2010	C18	80	27335.5
8:	Budapest	Férfi	75	2010	C18	75	22253.5
9:	Budapest	Férfi	80	2010	C18	72	15775.5
10:	Budapest	Férfi	85	2010	C18	35	10922.0
11:	Pest megye	Férfi	40	2010	C18	8	48086.0
12:	Pest megye	Férfi	45	2010	C18	6	36113.5
13:	Pest megye	Férfi	50	2010	C18	14	37167.0
14:	Pest megye	Férfi	55	2010	C18	32	41154.0
15:	Pest megye	Férfi	60	2010	C18	43	32527.0
16:	Pest megye	Férfi	65	2010	C18	56	26071.0
17:	Pest megye	Férfi	70	2010	C18	62	16926.5
18:	Pest megye	Férfi	75	2010	C18	45	11964.5
19:	Pest megye	Férfi	80	2010	C18	23	7015.0
20:	Pest megye	Férfi	85	2010	C18	11	4250.5
	County	Sex	Age	Year	ICDCode	N	Population

A kapott kód világosabb, gyorsabban beírható és jobban olvasható!

A data table azt is megengedi, hogy a vesszőt elhagyjuk (a data frame nem, ott hibát adna ha nem írnánk vesszőt!):

```
RawData[Year == 2010 & Age >= 40 &
  County %in% c("Budapest", "Pest megye") &
  Sex == "Férfi" & ICDCode == "C18"]
```

Key: <County, Sex, Age, Year>

	County	Sex	Age	Year	ICDCode	N	Population
	<char>	<char>	<num>	<num>	<char>	<int>	<num>
1:	Budapest	Férfi	40	2010	C18	3	57445.5
2:	Budapest	Férfi	45	2010	C18	10	42410.0
3:	Budapest	Férfi	50	2010	C18	17	45329.0
4:	Budapest	Férfi	55	2010	C18	44	55633.5
5:	Budapest	Férfi	60	2010	C18	59	45170.0
6:	Budapest	Férfi	65	2010	C18	120	39588.0
7:	Budapest	Férfi	70	2010	C18	80	27335.5
8:	Budapest	Férfi	75	2010	C18	75	22253.5
9:	Budapest	Férfi	80	2010	C18	72	15775.5
10:	Budapest	Férfi	85	2010	C18	35	10922.0
11:	Pest megye	Férfi	40	2010	C18	8	48086.0

12:	Pest megye	Férfi	45	2010	C18	6	36113.5
13:	Pest megye	Férfi	50	2010	C18	14	37167.0
14:	Pest megye	Férfi	55	2010	C18	32	41154.0
15:	Pest megye	Férfi	60	2010	C18	43	32527.0
16:	Pest megye	Férfi	65	2010	C18	56	26071.0
17:	Pest megye	Férfi	70	2010	C18	62	16926.5
18:	Pest megye	Férfi	75	2010	C18	45	11964.5
19:	Pest megye	Férfi	80	2010	C18	23	7015.0
20:	Pest megye	Férfi	85	2010	C18	11	4250.5
	County	Sex	Age	Year	ICDCode	N	Population

Fontos azonban, hogy ez csak ebben az esetben, tehát sorindexelésnél használható: ha nincs vessző, akkor automatikusan úgy veszi, hogy amit beírtunk, az sorindex (e megállapodás nélkül nem tudhatná, hogy mit akartunk indexelni).

A tény, hogy nem kell hivatkozni az adatkeret nevére, nem csak szűrésnél igaz, hanem rendezésnél is. Ezt ugyanis az `order` függvény valósítja meg, ami elérhető volt a data frame-hez is, csak ott ilyen módon kellett használnunk:

```
head(RawDataDF[order(RawDataDF$N),])
```

	County	Sex	Age	Year	ICDCode	N	Population
1	Baranya megye	Férfi	0	2000	C00	0	9876
2	Baranya megye	Férfi	0	2000	C01	0	9876
3	Baranya megye	Férfi	0	2000	C02	0	9876
4	Baranya megye	Férfi	0	2000	C03	0	9876
5	Baranya megye	Férfi	0	2000	C04	0	9876
6	Baranya megye	Férfi	0	2000	C05	0	9876

A data table azonban itt is megengedi¹¹ a fenti – nagyon logikus – egyszerűsítést:

```
RawData[order(N),]
```

	County	Sex	Age	Year	ICDCode	N	Population
	<char>	<char>	<num>	<num>	<char>	<int>	<num>
1:	Baranya megye	Férfi	0	2000	C00	0	9876.0

¹¹Egyébként ez utóbbi esetben nem ugyanaz az `order` fut le: a `data.table` definiál egy saját `order`-t, tehát az előbbi esetben a `base::order`, az utóbbinál a `data.table::order` fut. A `data.table` csomag `order`-je egyébként is okosabb, például sokkal kényelmesebb ha több változó szerint és változó irányban kell rendeznünk: egyszerűen fel kell sorolnunk az `order`-en belül a változókat, és amelyik szerint csökkenő sorrendben akarunk rendezni, ott ki kell tennünk a változó neve elé egy `-` jelet.

2:	Baranya megye	Férfi	0	2000	C01	0	9876.0
3:	Baranya megye	Férfi	0	2000	C02	0	9876.0
4:	Baranya megye	Férfi	0	2000	C03	0	9876.0
5:	Baranya megye	Férfi	0	2000	C04	0	9876.0

1313276:	Budapest	Nő	60	2008	C50	307	64674.0
1313277:	Budapest	Nő	55	2000	C50	308	67218.5
1313278:	Budapest	Nő	55	2002	C50	308	69118.0
1313279:	Budapest	Nő	70	2018	C44	331	56508.0
1313280:	Budapest	Nő	55	2003	C50	350	69396.5

Természetesen a „szűrés” és „rendezés” csak felhasználói szempontból két külön művelet. Az R számára a kettő ugyanaz: sorindexelés, csak annyi eltéréssel, hogy az előbbi esetben logikai vektort kap, az utóbbiban pedig számvektort (hiszen az `order` egyszerűen megadja sorban minden elemre, hogy az adott elem hányadik a nagyság szerinti sorrendben).

5.4. Kibővített oszlopindexelés: oszlop-kiválasztás és oszlop-létrehozás műveletekkel

Hagyományos data frame esetén a vessző után jön az oszlopindexelés, ami egy dolgot jelenthet: oszlopok kiválasztását. Tehát, dönthetünk, hogy mely oszlopokat kérjük (és melyeket nem), de más lehetőségünk nincs. Oszlopok kiválasztását célszerű mindig névvel és nem számmal végeznünk (hogy a kód az adatbázis esetleges későbbi módosításaira robusztusabb legyen, ne romoljon el új oszlop beszúrásától vagy törlésétől, valamint, hogy önállóan is jobban olvasható legyen a kód). Ekkor lényegében egy sztring-vektort kell átadnunk. A példa kedvéért itt – az előzőekkel szemben – a 40-45 éves budapesti férfiak vastagbélrákos eseteire szorítsuk meg magunkat, viszont tartsuk meg az összes évet. Ez esetben logikus csak az évet – és persze az `N`-et és a `Population`-t – kiíratni, hiszen a többi konstans:

```
head(RawDataDF[RawDataDF$Age == 40 &
  RawDataDF$County == "Budapest" &
  RawDataDF$Sex == "Férfi" &
  RawDataDF$ICDCode == "C18",
  c("Year", "N", "Population")])
```

	Year	N	Population
145939	2000	8	51602.0
146035	2001	4	47836.0
146131	2002	4	45296.5
146227	2003	4	43632.5

```
146323 2004 4      43085.0
146419 2005 5      43442.5
```

Ez a szintaktika a `data.table`-el is működik¹²:

```
RawData[Age == 40 & County == "Budapest" & Sex == "Férfi" &
        ICDCode == "C18",
        c("Year", "N", "Population")]
```

	Year	N	Population
	<num>	<int>	<num>
1:	2000	8	51602.0
2:	2001	4	47836.0
3:	2002	4	45296.5
4:	2003	4	43632.5
5:	2004	4	43085.0
6:	2005	5	43442.5
7:	2006	5	44511.5
8:	2007	6	46903.5
9:	2008	4	50505.5
10:	2009	6	54015.0
11:	2010	3	57445.5
12:	2011	8	60721.0
13:	2012	7	62471.5
14:	2013	5	63746.5
15:	2014	8	66250.5
16:	2015	13	70511.5
17:	2016	11	74622.0
18:	2017	6	77902.0
19:	2018	10	80555.0

A `data.table`-nek van azonban egy saját, külön szintaktikája erre, és célszerű is azt megszokni és használni mindig, mert a későbbi funkciókat az teszi elérhetővé:

¹²Valójában van egy különbség, ami akkor jelentkezik, ha a kiválasztandó oszlopok neveit eltávolítjuk egy változóban, és az indexelésnél ezt a változót szeretnénk felhasználni ahelyett, hogy kézzel beírjuk a neveket. Legyen például `colsel <- c("Year", "N", "Population")`. Ekkor a `data.frame`-nél mindegy, hogy a `RawDataDF[, c("Year", "N", "Population")]` vagy a `RawDataDF[, colsel]` formát használjuk, az eredmény ugyanaz lesz. Ami logikus is, hiszen látszólag ugyanazt írtuk be kétszer. Nagyon meglepő módon azonban a `data.table`-nél nem mindegy: a `RawData[, c("Year", "N", "Population")]` működni fog, de a `RawData[, colsel]` nem! Ennek az az oka, hogy `RawData[, colsel]` összeakad egy szintaktikával, amit később fogunk látni, és amelyben ez azt jelentené, hogy „válaszd ki a `colsel` nevű oszlopot és add vissza vektorként”. Ami természetesen nem fog sikerülni, hiszen ilyen nevű oszlop nincs. Van azonban megoldás: ha erre volna szükségünk akkor vagy a `RawData[, ..colsel]` vagy a `RawData[, colsel, with = FALSE]` alakot kell használnunk.

```
RawData[Age == 40 & County == "Budapest" & Sex == "Férfi" &
        ICDCCode == "C18",
        .(Year, N, Population)]
```

	Year	N	Population
	<num>	<int>	<num>
1:	2000	8	51602.0
2:	2001	4	47836.0
3:	2002	4	45296.5
4:	2003	4	43632.5
5:	2004	4	43085.0
6:	2005	5	43442.5
7:	2006	5	44511.5
8:	2007	6	46903.5
9:	2008	4	50505.5
10:	2009	6	54015.0
11:	2010	3	57445.5
12:	2011	8	60721.0
13:	2012	7	62471.5
14:	2013	5	63746.5
15:	2014	8	66250.5
16:	2015	13	70511.5
17:	2016	11	74622.0
18:	2017	6	77902.0
19:	2018	10	80555.0

Megjegyzendő, hogy a `.` egyszerűen egy rövidítés, amit a `data.table` csomag bevezet arra, hogy `list`, magyarul itt az történik, hogy egy listát kell átadnunk¹³, benne az – idézőjelek nélküli – oszlopnevekkel. A listás megoldás előnye, hogy valójában nem kötelező explicite kiírni, hogy `.` majd felsorolni a változóneveket zárójelben, bármilyen függvényt is használhatunk a vessző után ami listát ad eredményül. Később látunk majd erre példát.

Az is érthető a listás megoldás fényében, hogy `data table`-el átnevezhetünk változót úgymond „menet közben” (`data frame`-mel már ezt sem lehetett!):

```
RawData[Age == 40 & County == "Budapest" & Sex == "Férfi" &
        ICDCCode == "C18",
        .(Year, Esetszam = N, Lelekszam = Population)]
```

¹³Ez elsőre meglepő lehet, de valójában teljesen logikus: ha visszaemlékszünk, akkor már a `data.frame`-nél is láttuk, hogy az igazából az oszlopokból, mint vektorokból alkotott lista. Innen nézve teljesen érthető, hogy az oszlopokat egy lista elemeiként kell felsorolni!

	Year	Esetszam	Lelekszam
	<num>	<int>	<num>
1:	2000	8	51602.0
2:	2001	4	47836.0
3:	2002	4	45296.5
4:	2003	4	43632.5
5:	2004	4	43085.0
6:	2005	5	43442.5
7:	2006	5	44511.5
8:	2007	6	46903.5
9:	2008	4	50505.5
10:	2009	6	54015.0
11:	2010	3	57445.5
12:	2011	8	60721.0
13:	2012	7	62471.5
14:	2013	5	63746.5
15:	2014	8	66250.5
16:	2015	13	70511.5
17:	2016	11	74622.0
18:	2017	6	77902.0
19:	2018	10	80555.0

Ez már utat mutat a következő, igazi újdonsághoz.

Előtte még említsük meg, hogy a `data` table egyik jellegzetessége, hogy a `RawData[, .(Year)]` típusú hívások mindig `data` table-t adnak vissza¹⁴. Ha egyetlen változót választunk ki, de azt vektorként szeretnénk visszakapni (ez a kérdés nyilván csak egyetlen változó kiválasztásakor merül fel), akkor használjuk a `RawData$Year` vagy a `RawData[["Year"]]` formát¹⁵.

Ez eddig nem nagy változás, még csak azt sem igazán lehet mondani, hogy az előzőhöz hasonló kényelmi továbbfejlesztés, hiszen ez a szintaktika nem sokkal tér el a korábbtól. Az igazán érdekes rész azonban most jön, a `data.table` ugyanis lehetővé tesz valamit, ami a `data.frame`-nél fel sem merült: nem csak passzívan kiválaszthatunk oszlopokat, hanem *műveleteket is végezhetünk* velük, így új oszlopokat hozva létre! Lényegében „on the fly”, azaz menet közben végezhetünk műveleteket és hozhatunk létre új oszlopokat, anélkül, hogy azokat fizikailag le kellene tárolnunk az adatbázisba. A `data` table vessző utáni pozíciójában tehát

¹⁴Ez nem nyilvánvaló: a `RawDataDF[, "Year"]` egy vektor lesz! Természetesen a `RawDataDF[, c("Year", "County")]` megint csak `data` frame; vagyis lényegében az történik, hogy a `data` frame automatikusan egyszerűsít: ha lehet – azaz egyetlen változót (oszlopot) választottunk ki – akkor egyszerűsíti vektorrá, ha nem, mert többet, akkor marad a `data` frame. Ez kényelmes is lehet, de közben mégis csak egy inkonzisztencia, hogy ugyanolyan típusú hívások eredménye teljesen eltérő adatstruktúra is lehet. Ezzel szemben a `data.table`-nél a `RawData[, .(...)]` típusú hívások *mindig* `data` table-t adnak vissza.

¹⁵Elvileg a `RawData[, Year]` is használható, de ezt talán jobb kerülni, ritkán fordul elő.

Például a rákos megbetegedéseknél fontos az incidencia, tehát a lélekszámhoz viszonyított előfordulás. (Értelemszerűen nem mindegy, hogy 10 vagy 10 ezer ember közül került ki 1 rákos adott évben.) Ezt tipikusan 100 ezer lakosra vonatkoztatva szokták megadni. Nézzük meg a következő data table-t használó megoldást:

```
RawData[Age == 40 & County == "Budapest" & Sex == "Férfi" &
  ICDCode == "C18",
  .(Year, N, Population, Inc = N / Population * 1e5)]
```

	Year	N	Population	Inc
	<num>	<int>	<num>	<num>
1:	2000	8	51602.0	15.503275
2:	2001	4	47836.0	8.361903
3:	2002	4	45296.5	8.830704
4:	2003	4	43632.5	9.167478
5:	2004	4	43085.0	9.283974
6:	2005	5	43442.5	11.509467
7:	2006	5	44511.5	11.233052
8:	2007	6	46903.5	12.792222
9:	2008	4	50505.5	7.919930
10:	2009	6	54015.0	11.108026
11:	2010	3	57445.5	5.222341
12:	2011	8	60721.0	13.175014
13:	2012	7	62471.5	11.205110
14:	2013	5	63746.5	7.843568
15:	2014	8	66250.5	12.075381
16:	2015	13	70511.5	18.436709
17:	2016	11	74622.0	14.740961
18:	2017	6	77902.0	7.701985
19:	2018	10	80555.0	12.413879

Azaz az Inc oszlopot létrehoztuk a nélkül, hogy előzetesen azt le kellett volna tárolnunk magába az adatbázisba! Menet közben számoltuk ki, és még nevet is adtunk neki. Az oszlopok tehát itt, a vessző utáni pozícióban úgy viselkednek egy data table-nél mintha szokásos változók lennének!

Ebből is adódik, hogy a lehetőségeink még ennél is bővebbek: nem csak egyszerű aritmetikai műveleteket végezhetünk egy oszloppal (vagy épp több oszloppal! – mint arra ez előbbi kód is példát mutat), hanem *bármilyen* R függvényt rájuk ereszthetünk! Tekintsünk példának a következő kódot, mely megadja, hogy a 40-45 éves budapesti férfiak körében összesen hány vastagbélrákos eset volt az adatbázis által lefedett 19 év alatt:

```
RawData[Age == 40 & County == "Budapest" & Sex == "Férfi" &
  ICDCode == "C18",
  .(N = sum(N))]
```

```
      N
<int>
1: 121
```

Az N oszlop egy vektor, tehát azon túl, hogy oszthatjuk – elemenként – egy másik vektorral, mint ahogy az előbbi esetben tettük, nyugodtan összegezhethetjük is példának okáért. Ebből mellesleg az is látszik, hogy még az sem jelent problémát, ha a művelet által visszaadott eredménynek a hossza is eltér a bemenő változóétól! Hiszen a `sum(N)` 1 hosszú, míg a `Year` 19. (Az azonban fontos, hogy itt már a `Year` nem szerepel a kiválasztott oszlopok között: megtarthattuk volna a `Year`-t is, de mivel az 19 hosszú, így a mellette lévő oszlopban ugyanaz az összeg 19-szer meg lett volna ismételve.)

A fenti példákban egyszerre szűrtünk sorokat és számoltunk oszlopokat. (Ez természetesen nem kötelező, lehet csak az egyiket csinálni a másik nélkül.) Egyetlen példa a `data.table` optimalizálására: ilyenkor nem azt csinálja, hogy leszűri az egész adatbázist, és aztán végzi az oszlopműveleteket, hanem először megnézi, hogy mely oszlopokra van egyáltalán szükség – például csak a `Year`-re, `N`-re és `Population`-re – és ilyenkor *csak* azokat szűri le, így kerülve el, hogy olyan oszlopok szűrését is el kelljen végeznie, amik később nem is jelennek meg az eredményben. Ez azért lehetséges, mert a `data.table` „egyben látja” az egész feladatot, és így tud ilyen optimalizálásokat tenni.

Visszatérve, a dolog még jobban kombinálható: legyen a példa kedvéért a feladatunk az, hogy számoljuk ki az egész 19 éves periódusra az incidenciát. (Egy pillanatra érdemes itt megállni, és végiggondolni, hogy mi egyáltalán az ehhez szükséges művelet!) Íme a megvalósítás `data.table` használatával:

```
RawData[Age == 40 & County == "Budapest" & Sex == "Férfi" &
  ICDCode == "C18",
  .(Inc = sum(N) / sum(Population) * 1e5)]
```

```
      Inc
<num>
1: 11.1515
```

Amint láthatjuk, tetszőleges komplexitású műveletet, számítást elvégezhetünk a vessző után! És ezt szó szerint kell érteni: *bármilyen* R függvényt használhatunk az oszlopindexelés pozíciójában, a vessző után, bármilyen műveletet vagy számítást végezhetünk (tehát még csak olyan megkötés

sincs, hogy csak bizonyos függvényeket, műveleteket tesz csak elérhetővé a `data.table`). Íme egy példa; lognormális eloszlást illesztünk az esetszámok különböző években mért értékeiből kapott eloszlásra:

```
RawData[Age == 40 & County == "Budapest" & Sex == "Férfi" &
  ICDCCode == "C18",
  .(fitdistrplus::fitdist(N, "lnorm")$estimate["meanlog"])]
```

```
      V1
      <num>
1: 1.772807
```

Szépen látszik itt is, hogy nyugodtan használhatjuk az `N`-et csak így, minden további nélkül – ugyanúgy viselkedik, mint egy szokásos változó, ugyanúgy használhatjuk egy számítás során.

Ráadásul, ha visszaemlékszünk, akkor szerepelt, hogy a vessző utáni pozícióban egy listának kell szerepelnie – de ezt előállíthatja egy függvény is! Például a `fitdistrplus::fitdist` eredményének `estimate` nevű komponense egy vektor. De ha ez `as.list`-tel átalakítjuk, akkor egy listát kapunk, így közvetlenül átadható a vessző utáni pozícióban (természetesen ilyenkor `.` nem kell, hiszen az `as.list` *elvé* egy listát ad vissza!):

```
RawData[Age == 40 & County == "Budapest" & Sex == "Férfi" &
  ICDCCode == "C18",
  as.list(fitdistrplus::fitdist(N, "lnorm")$estimate)]
```

```
      meanlog      sdlog
      <num>      <num>
1: 1.772807 0.3902478
```

Ez tehát már messze-messze nem csak egyszerűen oszlop kiválasztás, amire itt módunk van, ha `data.table`-t használunk.

Egyetlen megjegyzés a végére: mi van akkor, ha kíváncsiak vagyunk arra, hogy hány sor van egy adattáblában (esetleg szűkítés után)? A `RawData[Age == 40 & County == "Budapest" & Sex == "Férfi" & ICDCCode == "C18", length(N)]` kézenfekvő megoldás, de nem túl elegáns (miért pont az `N` hosszát néztük meg? bármi más is ugyanezt az eredményt adná!). Erre a célra a `data.table` bevezet egy speciális szimbólumot, a `.N`-et, ami egyszerűen visszaadja¹⁶ a sorok számát:

¹⁶Észrevehető, hogy az eredmény egy szám lesz, nem egy data table. Ennek az oka, hogy a `.N` – hiába van a nevében egy `.` – nem egy lista. Ha data frame-et szeretnénk visszakapni, akkor a korábbiakkal összhangban azt kell írunk, hogy `.(.N)`.

```
RawData[Age == 40 & County == "Budapest" & Sex == "Férfi" &
        ICDCode == "C18", .N]
```

[1] 19

5.5. Csoportosítás (aggregáció)

A `data.table` második, rendkívül erőteljes bővítése a hagyományos data frame funkcionálisához képest a csoportosítás (aggregáció) lehetősége. A `data.table` bevezet egy *harmadik* pozíciót a szögletes zárójelen belül: megtehetjük, hogy *két* vesszőt teszünk ki a szögletes zárójelen belül, ez esetben az első vessző előtt van a sorindexelés (ahogy eddig is), az első és a második vessző között az oszlopkiválasztás és -számítás (ahogy eddig is), viszont a második vessző *után* megadhatunk egy listát egy vagy több változóból. (A `.` ugyanúgy használható a `list` helyett. Megadhatunk sztring-vektort is, benne a változók neveivel; ez különösen jól jön akkor, ha gépi úton állítjuk elő, hogy mik ezek a változók.) Mi fog ilyenkor történni? A `data.table` elsőként végrehajtja a sorok szűrését, ha kértünk ilyet, ezután pedig az új, harmadik pozícióban megadott változó vagy változók szerint csoportokat képez. Mit jelent az, hogy „csoport”? Azok a sorai a táblának, amelyekben a csoportosító változó egy adott értéket vesz fel: ahány lehetséges értéke van a csoportosító változónak a táblában, annyi csoport képződik, úgy, hogy csoporton belül a csoportosító változó homogén lesz. Ezt követően a `data.table` végrehajtja a megadott oszlopkiválasztásokat és/vagy oszlopműveleteket csoportonként *külön-külön*, végül pedig a kapott eredményeket újra összerakja egy táblába, úgy, hogy mindegyik csoport eredménye mellé beteszi oszlopként azt, hogy ott mi volt a csoportosító változó értéke. Az egyes csoportok abban a sorrendben fognak szerepelni az eredményben, ahogy egymás után jöttek a kiinduló táblában.

A jobb megértés kedvéért nézzünk egy gyakorlati példát! Kíváncsiak vagyunk az egész időintervallumra vonatkozó incidenciára, de az *összes* rák-típus esetén külön-külön megadva. Mit tudunk tenni? Fent láttuk a kódot, mely egy adott típusra ezt kiszámolja. Az remélhetőleg senkinek nem jut az eszébe, hogy kézzel lefuttassa először C00-val, aztán C01-gyel, aztán C02-vel... Működőképesebb megoldás ennek valamilyen R paranccsal történő automatizálása. Rosszabb esetben a `for` jut az eszünkbe, jobb esetben az `apply` család valamely tagja. (A `for`-ciklus rosszabb eset, mert az R-ben a legtöbb esetben illendő kerülni, és jelen esetben tényleg meg is oldható a probléma megfelelő `apply` használatával, így ez is a célszerű választás.) Ha azonban a `data.table`-t használjuk, akkor még csak erre sincs szükség!

Nézzük ugyanis meg a következő hívást:

```
RawData[Age == 40 & County == "Budapest" & Sex == "Férfi",
        .(Inc = sum(N) / sum(Population) * 1e5),
        .(ICDCode)]
```

	ICDCode	Inc
	<char>	<num>
1:	C00	0.36864474
2:	C01	2.21186843
3:	C02	2.58051316
4:	C03	1.01377303
5:	C04	2.48835198
6:	C05	0.73728948
7:	C06	0.55296711
8:	C07	0.92161184
9:	C08	0.73728948
10:	C09	1.65890132
11:	C10	3.13348027
12:	C11	1.19809540
13:	C12	0.46080592
14:	C13	4.42373685
15:	C14	0.82945066
16:	C15	4.14725330
17:	C16	6.72776646
18:	C17	1.75106250
19:	C18	11.15150331
20:	C19	2.58051316
21:	C20	7.09641120
22:	C21	0.55296711
23:	C22	3.87076974
24:	C23	1.01377303
25:	C24	1.47457895
26:	C25	7.37289475
27:	C26	0.46080592
28:	C30	0.64512829
29:	C31	1.10593421
30:	C32	7.46505593
31:	C33	0.09216118
32:	C34	30.78183558
33:	C37	0.46080592
34:	C38	1.10593421
35:	C39	0.36864474
36:	C40	1.01377303
37:	C41	2.94915790
38:	C43	17.41846385
39:	C44	43.13143429
40:	C45	0.64512829
41:	C46	0.46080592

42:	C47	0.00000000
43:	C48	2.39619079
44:	C49	10.78285857
45:	C50	2.76483553
46:	C51	0.00000000
47:	C52	0.00000000
48:	C53	0.00000000
49:	C54	0.00000000
50:	C55	0.00000000
51:	C56	0.00000000
52:	C57	0.00000000
53:	C58	0.00000000
54:	C60	0.92161184
55:	C61	2.94915790
56:	C62	22.11868425
57:	C63	0.64512829
58:	C64	12.25743752
59:	C65	0.27648355
60:	C66	0.00000000
61:	C67	8.38666778
62:	C68	0.18432237
63:	C69	1.10593421
64:	C70	0.55296711
65:	C71	8.75531252
66:	C72	1.10593421
67:	C73	5.06886514
68:	C74	0.73728948
69:	C75	0.27648355
70:	C76	2.67267435
71:	C80	2.39619079
72:	C81	3.40996382
73:	C82	2.58051316
74:	C83	3.96293093
75:	C84	0.92161184
76:	C85	4.42373685
77:	C88	0.00000000
78:	C90	2.21186843
79:	C91	3.31780264
80:	C92	4.51589803
81:	C93	0.00000000
82:	C94	0.18432237
83:	C95	0.09216118
84:	C96	1.56674013

85:	C97	0.00000000
86:	D00	0.46080592
87:	D01	0.18432237
88:	D02	0.09216118
89:	D03	4.60805922
90:	D04	1.75106250
91:	D05	0.00000000
92:	D06	0.00000000
93:	D07	0.36864474
94:	D09	0.27648355
95:	D30	1.38241777
96:	D33	6.26696054
	ICDCode	Inc

Mi történt itt? Először is, a sor-szűrések közül kivettük a konkrét rák-típust – ez értelemszerű, hiszen az összes ráktípusra vonatkozó adatot szeretnénk kapni, épp ez volt a feladat, tehát ebben nyilván nem szűrhetjük le előzetesen az adatbázist. Másodszor, bekerült a harmadik pozíciója, csoportosító változóként a rák típusa. Mit jelent ez? Azt, hogy a szűrés után a `data.table` a leszűrt adatbázisból rák-típus szerint csoportokat képez, tehát szétszedi az adatbázist kis táblákra úgy, hogy mindegyikben egy adott rák-típus adatai legyenek, mindegyikre elvégzi a második pozícióban, az oszlopindexelésnél megadott műveleteket (jelen esetben: kiszámítja az incidenciákat), majd ezeket az eredményeket, ami itt most egyetlen sor lesz, újra összerakja egy nagy táblába, jelezve, hogy az adott eredmény melyik kódhoz tartozik.

Nagyon szájbarágós, de talán egyszer érdemes a dolgot megnézni lépésről-lépésre. A `data.table` elsőként leszűri a táblát a sorindex szerint:

```
RawData[Age == 40 & County == "Budapest" & Sex == "Férfi"]
```

Key: <County, Sex, Age, Year>

	County	Sex	Age	Year	ICDCode	N	Population
	<char>	<char>	<num>	<num>	<char>	<int>	<num>
1:	Budapest	Férfi	40	2000	C00	1	51602
2:	Budapest	Férfi	40	2000	C01	6	51602
3:	Budapest	Férfi	40	2000	C02	2	51602
4:	Budapest	Férfi	40	2000	C03	1	51602
5:	Budapest	Férfi	40	2000	C04	2	51602

1820:	Budapest	Férfi	40	2018	D06	0	80555
1821:	Budapest	Férfi	40	2018	D07	0	80555
1822:	Budapest	Férfi	40	2018	D09	2	80555
1823:	Budapest	Férfi	40	2018	D30	0	80555
1824:	Budapest	Férfi	40	2018	D33	4	80555

Ezt követően megnézi, hogy a csoportosító változó milyen értékeket vesz fel:

```
unique(RawData[Age == 40 & County == "Budapest" &
          Sex == "Férfi"]$ICDCode)
```

```
[1] "C00" "C01" "C02" "C03" "C04" "C05" "C06" "C07" "C08" "C09" "C10" "C11"
[13] "C12" "C13" "C14" "C15" "C16" "C17" "C18" "C19" "C20" "C21" "C22" "C23"
[25] "C24" "C25" "C26" "C30" "C31" "C32" "C33" "C34" "C37" "C38" "C39" "C40"
[37] "C41" "C43" "C44" "C45" "C46" "C47" "C48" "C49" "C50" "C51" "C52" "C53"
[49] "C54" "C55" "C56" "C57" "C58" "C60" "C61" "C62" "C63" "C64" "C65" "C66"
[61] "C67" "C68" "C69" "C70" "C71" "C72" "C73" "C74" "C75" "C76" "C80" "C81"
[73] "C82" "C83" "C84" "C85" "C88" "C90" "C91" "C92" "C93" "C94" "C95" "C96"
[85] "C97" "D00" "D01" "D02" "D03" "D04" "D05" "D06" "D07" "D09" "D30" "D33"
```

Majd ezek mindegyikére leszűkíti a (szűrt) táblát, lényegében kis táblákat készítve. Így néz ki a C00-hoz tartozó:

```
RawData[Age == 40 & County == "Budapest" & Sex == "Férfi" &
          ICDCode == "C00"]
```

Key: <County, Sex, Age, Year>

	County	Sex	Age	Year	ICDCode	N	Population
	<char>	<char>	<num>	<num>	<char>	<int>	<num>
1:	Budapest	Férfi	40	2000	C00	1	51602.0
2:	Budapest	Férfi	40	2001	C00	1	47836.0
3:	Budapest	Férfi	40	2002	C00	0	45296.5
4:	Budapest	Férfi	40	2003	C00	0	43632.5
5:	Budapest	Férfi	40	2004	C00	0	43085.0
6:	Budapest	Férfi	40	2005	C00	0	43442.5
7:	Budapest	Férfi	40	2006	C00	0	44511.5
8:	Budapest	Férfi	40	2007	C00	0	46903.5
9:	Budapest	Férfi	40	2008	C00	0	50505.5
10:	Budapest	Férfi	40	2009	C00	0	54015.0
11:	Budapest	Férfi	40	2010	C00	0	57445.5
12:	Budapest	Férfi	40	2011	C00	1	60721.0
13:	Budapest	Férfi	40	2012	C00	1	62471.5
14:	Budapest	Férfi	40	2013	C00	0	63746.5
15:	Budapest	Férfi	40	2014	C00	0	66250.5
16:	Budapest	Férfi	40	2015	C00	0	70511.5
17:	Budapest	Férfi	40	2016	C00	0	74622.0
18:	Budapest	Férfi	40	2017	C00	0	77902.0
19:	Budapest	Férfi	40	2018	C00	0	80555.0

Így a C01-hez:

```
RawData[Age == 40 & County == "Budapest" & Sex == "Férfi" &
        ICDCCode == "C01"]
```

Key: <County, Sex, Age, Year>

	County	Sex	Age	Year	ICDCCode	N	Population
	<char>	<char>	<num>	<num>	<char>	<int>	<num>
1:	Budapest	Férfi	40	2000	C01	6	51602.0
2:	Budapest	Férfi	40	2001	C01	1	47836.0
3:	Budapest	Férfi	40	2002	C01	0	45296.5
4:	Budapest	Férfi	40	2003	C01	3	43632.5
5:	Budapest	Férfi	40	2004	C01	2	43085.0
6:	Budapest	Férfi	40	2005	C01	0	43442.5
7:	Budapest	Férfi	40	2006	C01	3	44511.5
8:	Budapest	Férfi	40	2007	C01	2	46903.5
9:	Budapest	Férfi	40	2008	C01	0	50505.5
10:	Budapest	Férfi	40	2009	C01	2	54015.0
11:	Budapest	Férfi	40	2010	C01	0	57445.5
12:	Budapest	Férfi	40	2011	C01	3	60721.0
13:	Budapest	Férfi	40	2012	C01	1	62471.5
14:	Budapest	Férfi	40	2013	C01	1	63746.5
15:	Budapest	Férfi	40	2014	C01	0	66250.5
16:	Budapest	Férfi	40	2015	C01	0	70511.5
17:	Budapest	Férfi	40	2016	C01	0	74622.0
18:	Budapest	Férfi	40	2017	C01	0	77902.0
19:	Budapest	Férfi	40	2018	C01	0	80555.0

És így tovább.

Ezt követően minden kis táblára elvégzi az oszlopindexelésnél kijelölt műveletet. Így fog kinézni az eredmény a C00-s kis táblára:

```
RawData[Age == 40 & County == "Budapest" & Sex == "Férfi" &
        ICDCCode == "C00",
        .(Inc = sum(N) / sum(Population) * 1e5)]
```

```
Inc
<num>
1: 0.3686447
```

Így a C01-es kis táblára:

```
RawData[Age == 40 & County == "Budapest" & Sex == "Férfi" &
  ICDCode == "C01",
  .(Inc = sum(N) / sum(Population) * 1e5)]
```

```
      Inc
<num>
1: 2.211868
```

Majd ezeket a kis táblákat egymás alá rendezi, abban a sorrendben, ahogy az eredeti táblában előfordultak a csoportosító változó értékei, és úgy, hogy mindegyikhez melléírja, hogy az adottnál mi volt a csoportosító változó értéke, tehát jelen esetben, hogy melyik ráktípushoz tartozik.

Így kaptuk a fent látható táblát (menjünk vissza és ellenőrizzük)...!

Nézzünk meg – most már nagyon részletes levezetés nélkül – még egy példát csoportosításra. Kíváncsiak vagyunk egy adott ráktípus korszpecifikus incidenciájára, tehát, hogy mennyi az incidencia adott életkorban. Ha mindezt rögzített évre, nemre és megyére kérdezzük, akkor célt érhetünk így:

```
RawData[Year == 2010 & County == "Budapest" &
  Sex == "Férfi" & ICDCode == "C18",
  .(Age, Inc = N / Population * 1e5)]
```

	Age	Inc
	<num>	<num>
1:	0	0.000000
2:	5	0.000000
3:	10	0.000000
4:	15	0.000000
5:	20	1.960054
6:	25	3.072716
7:	30	2.321088
8:	35	11.088472
9:	40	5.222341
10:	45	23.579344
11:	50	37.503585
12:	55	79.089038
13:	60	130.617667
14:	65	303.122158


```

15:    70 292.659728
16:    75 337.025636
17:    80 456.403917
18:    85 320.454129

```

A dolog azonban nagyon nem szerencsés: kizárólag azért fog működni, mert a leszűkítés után egy adott életkorhoz már csak egyetlen sor tartozik. De ha ez nem így lenne, például kitörlünk valamit a feltételek közül, akkor teljesen rossz eredményt fog adni, hiszen ilyenkor ugyanaz az életkor többször fog megjelenni az eredményben, míg nekünk össze kellene adnunk az adott életkorhoz tartozó különböző megfigyeléseket.

A megoldás a csoportosítás az életkor szerint, és az összeadás adott életkoron belül:

```

RawData[Year == 2010 & County == "Budapest" &
  Sex == "Férfi" & ICDCode == "C18",
  .(Inc = sum(N) / sum(Population) * 1e5), .(Age)]

```

	Age	Inc
	<num>	<num>
1:	0	0.000000
2:	5	0.000000
3:	10	0.000000
4:	15	0.000000
5:	20	1.960054
6:	25	3.072716
7:	30	2.321088
8:	35	11.088472
9:	40	5.222341
10:	45	23.579344
11:	50	37.503585
12:	55	79.089038
13:	60	130.617667
14:	65	303.122158
15:	70	292.659728
16:	75	337.025636
17:	80	456.403917
18:	85	320.454129

Ez immár működik másféle szűréssel is, például ha Budapest helyett az egész országra vagyunk kíváncsiak:

```
RawData[Year == 2010 & Sex == "Férfi" & ICDCode == "C18",
        .(Inc = sum(N) / sum(Population) * 1e5), .(Age)]
```

	Age	Inc
	<num>	<num>
1:	0	0.4013437
2:	5	0.0000000
3:	10	0.3909763
4:	15	0.3278087
5:	20	1.2121488
6:	25	2.2652654
7:	30	3.2732344
8:	35	7.1454000
9:	40	11.2810011
10:	45	21.7613797
11:	50	46.2794518
12:	55	94.8719665
13:	60	130.8683356
14:	65	246.5976935
15:	70	293.1432072
16:	75	367.2073711
17:	80	347.1158754
18:	85	323.4103513

Érdemes végiggondolni (ez általában is hasznos): ilyenkor az életkor szerinti kis táblákban 20 sor lesz – az egyes megyékkel – és ezek fölött fogunk összegezni.

Megjegyzendő, hogy a csoportosító változónak nevet is adhatunk:

```
RawData[Year == 2010 & Sex == "Férfi" & ICDCode == "C18",
        .(Inc = sum(N) / sum(Population) * 1e5),
        .(Eletkor = Age)]
```

	Eletkor	Inc
	<num>	<num>
1:	0	0.4013437
2:	5	0.0000000
3:	10	0.3909763
4:	15	0.3278087
5:	20	1.2121488
6:	25	2.2652654

```

7:      30    3.2732344
8:      35    7.1454000
9:      40   11.2810011
10:     45   21.7613797
11:     50   46.2794518
12:     55   94.8719665
13:     60  130.8683356
14:     65  246.5976935
15:     70  293.1432072
16:     75  367.2073711
17:     80  347.1158754
18:     85  323.4103513

```

Ami azonban sokkal izgalmasabb, hogy műveletet is végezhetünk! Itt is igaz, hogy nem kell a változót külön letárolni, hanem menet közben kiszámolhatjuk, majd építhetünk is rá rögtön (jelen esetben egy csoportosítást). Például, ha ki akarjuk számolni az incidenciát külön a 70 év alattiak és felettiak körében:

```

RawData[Year == 2010 & Sex == "Férfi" & ICDCode == "C18",
        .(Inc = sum(N) / sum(Population) * 1e5),
        .(Idos = Age > 70)]

```

```

      Idos      Inc
<lgcl> <num>
1: FALSE  43.83737
2:  TRUE 352.54419

```

5.6. Indexelések láncolása egymás után

A `data.table` következő újítása, hogy megengedi egy már indexelt tábla (`RawData[...]`) *újabb* indexelését. Tehát használhatjuk a `RawData[...][...]` alakot, ahol a második indexelés pontosan ugyanúgy fog viselkedni, mint az első (ugyanúgy használhatunk sorindexelést, szűrést és rendezést, oszlopkiválasztást és -transzformációt, csoportosítást), *de* úgy, hogy az az első, már indexelt táblára vonatkozik! Lényegében mintha elmentettük volna a `RawData[...]`-t egy változóba, és utána azt a változót indexelnénk szokásos módon – csak itt nem kell semmit külön elmenteni. Az, hogy a második indexelés már az első indexelésben átalakított táblára vonatkozik, egy kritikusan fontos előny, amint az rögtön világossá is fog válni.

Ha pontosak akarunk lenni, akkor ezt az egymás utáni többszöri indexelést igazából a hagyományos data frame is megengedi, tehát például a `RawDataDF[101:200,][5:15,]` egy teljesen szabályos hívás (és természetesen egyenértékű lesz azzal, hogy `RawDataDF[105:115,]`). A

probléma az, hogy a használhatósága nagyon korlátozott, mert a második indexben, ha változóra hivatkozunk, az az *eredeti* adatkeret változója tud csak lenni, nem az első indexelésben már áttanszformálté! (Értelemszerűen, hiszen az nincs is elmentve, nincs is semmilyen külön neve, ahogy hivatkozhatnánk rá.) Ha csak a legegyszerűbb transzformációt, a sorok szűrését vesszük: a `RawDataDF[RawDataDF$Sex == "Férfi",][RawDataDF$Year == 2010,]` nem fog működni, ez onnan is kapásból látszik, hogy a `RawDataDF$Year == 2010` ugyanolyan hosszú, mint a `RawDataDF`, viszont a `RawDataDF[RawDataDF$Sex == "Férfi",]` már rövidebb, tehát ez így biztosan nem lehet jó, mert az adattáblát hosszabb vektorral próbáljuk indexelni, mint ahány sora van. Data frame használatával erre a problémára nincs megoldás, hiszen a `RawDataDF[RawDataDF$Sex == "Férfi",]` táblázat `Year` változójára nem tudunk sehogy sem hivatkozni a második indexelésben, hiszen az nincs elmentve, nincs is külön neve, amivel hivatkozhatnánk.

A data table esetében azonban, kihasználva, hogy a változóra hivatkozhatunk csak a nevével, a táblázat neve nélkül, erre nagyon egyszerű a megoldás: annyi a feladat, hogy a második indexben szereplő `Year` alatt a `data.table` azt értse, hogy az első indexelés *után* kapott táblázat `Year` nevű változója (ne azt, hogy az eredetié). És így is van megírva a `data.table`, ezért szerepelt korábban az a megfogalmazás, hogy a második index az első indexeléssel már transzformált táblára vonatkozik. Így aztán a következő hívás tökéletesen működik data table-lel:

```
RawData[Sex == "Férfi"][Year == 2010]
```

Key: <County, Sex, Age, Year>

	County	Sex	Age	Year	ICDCode	N	Population
	<char>	<char>	<num>	<num>	<char>	<int>	<num>
1:	Baranya megye	Férfi	0	2010	C00	0	9430.0
2:	Baranya megye	Férfi	0	2010	C01	0	9430.0
3:	Baranya megye	Férfi	0	2010	C02	0	9430.0
4:	Baranya megye	Férfi	0	2010	C03	0	9430.0
5:	Baranya megye	Férfi	0	2010	C04	0	9430.0

34556:	Zala megye	Férfi	85	2010	D06	0	1447.5
34557:	Zala megye	Férfi	85	2010	D07	0	1447.5
34558:	Zala megye	Férfi	85	2010	D09	0	1447.5
34559:	Zala megye	Férfi	85	2010	D30	0	1447.5
34560:	Zala megye	Férfi	85	2010	D33	4	1447.5

Ez még nem a legátütőbb példa – bár sokszor az ilyenek is nagyon jól jönnek – hiszen használhattunk volna egyszerűen `&` jelet és egyetlen indexelést. A dolog igazi erejét az adja, hogy – ismét csak abból fakadóan, hogy a második index már az elsőnek indexelt táblát látja, neki nem is számít, hogy az nem egy lementett tábla, hanem egy már átalakított – módunk van menet közben létrehozott változókra is hivatkozni! Például miután kiszámoltuk rák-típusonként

az incidenciát, szeretnénk a táblázatot az incidenciák szerint növekvő sorba rakni. Íme a megoldás:

```
RawData[Age == 40 & County == "Budapest" & Sex == "Férfi",
        .(Inc = sum(N) / sum(Population) * 1e5),
        .(ICDCode)][order(Inc)]
```

	ICDCode	Inc
	<char>	<num>
1:	C47	0.00000000
2:	C51	0.00000000
3:	C52	0.00000000
4:	C53	0.00000000
5:	C54	0.00000000
6:	C55	0.00000000
7:	C56	0.00000000
8:	C57	0.00000000
9:	C58	0.00000000
10:	C66	0.00000000
11:	C88	0.00000000
12:	C93	0.00000000
13:	C97	0.00000000
14:	D05	0.00000000
15:	D06	0.00000000
16:	C33	0.09216118
17:	C95	0.09216118
18:	D02	0.09216118
19:	C68	0.18432237
20:	C94	0.18432237
21:	D01	0.18432237
22:	C65	0.27648355
23:	C75	0.27648355
24:	D09	0.27648355
25:	C00	0.36864474
26:	C39	0.36864474
27:	D07	0.36864474
28:	C12	0.46080592
29:	C26	0.46080592
30:	C37	0.46080592
31:	C46	0.46080592
32:	D00	0.46080592
33:	C06	0.55296711

34:	C21	0.55296711
35:	C70	0.55296711
36:	C30	0.64512829
37:	C45	0.64512829
38:	C63	0.64512829
39:	C05	0.73728948
40:	C08	0.73728948
41:	C74	0.73728948
42:	C14	0.82945066
43:	C07	0.92161184
44:	C60	0.92161184
45:	C84	0.92161184
46:	C03	1.01377303
47:	C23	1.01377303
48:	C40	1.01377303
49:	C31	1.10593421
50:	C38	1.10593421
51:	C69	1.10593421
52:	C72	1.10593421
53:	C11	1.19809540
54:	D30	1.38241777
55:	C24	1.47457895
56:	C96	1.56674013
57:	C09	1.65890132
58:	C17	1.75106250
59:	D04	1.75106250
60:	C01	2.21186843
61:	C90	2.21186843
62:	C48	2.39619079
63:	C80	2.39619079
64:	C04	2.48835198
65:	C02	2.58051316
66:	C19	2.58051316
67:	C82	2.58051316
68:	C76	2.67267435
69:	C50	2.76483553
70:	C41	2.94915790
71:	C61	2.94915790
72:	C10	3.13348027
73:	C91	3.31780264
74:	C81	3.40996382
75:	C22	3.87076974
76:	C83	3.96293093

77:	C15	4.14725330
78:	C13	4.42373685
79:	C85	4.42373685
80:	C92	4.51589803
81:	D03	4.60805922
82:	C73	5.06886514
83:	D33	6.26696054
84:	C16	6.72776646
85:	C20	7.09641120
86:	C25	7.37289475
87:	C32	7.46505593
88:	C67	8.38666778
89:	C71	8.75531252
90:	C49	10.78285857
91:	C18	11.15150331
92:	C64	12.25743752
93:	C43	17.41846385
94:	C62	22.11868425
95:	C34	30.78183558
96:	C44	43.13143429
	ICDCode	Inc

Hiába nem is létezik `Inc` nevű változó az eredeti adattáblában, ez a hívás mégis tökéletesen fog működni! Megint csak: azért, mert a második index már az első indexeléssel átalakított táblát kapja meg, és azt látja, pontosan ugyanúgy, mintha az egy lementett tábla lenne.

5.7. Referencia szemantika

A `data.table` bevezet egy új megközelítést arra, hogy új változót definiáljunk egy táblában – ám hamar ki fog derülni, hogy itt jóval többről van szó, mint egyszerűen egy alternatív jelölésről.

Például számoljuk ki, és ezúttal a táblázatban is tároljuk el az incidenciákat¹⁷:

```
RawData$Inc <- RawData$N / RawData$Population * 1e5
```

¹⁷Az összes fenti esetben ezt el tudtuk kerülni, és jobb is elkerülni: gondoljunk arra, hogy ha csoportosítást is csinálunk, akkor ezekkel az előre kiszámolt rétegenkénti incidenciákkal nem megyünk semmire. (Általában is igaz, hogy a kiszámítható dolgok közül csak azokat érdemes fizikailag letárolni az adatbázisban, amik kiszámítása sok időt venne igénybe.) Tehát ez most szigorúan csak illusztratív példa új változó létrehozására.

A `data.table` által bevezett új megoldás esetén az értékadás jele a `:=`, de ami talán még fontosabb, hogy ezt, elsőre elég meglepő módon, úgy kell megadni, mintha indexelnénk, tehát szögletes zárójelek között! A második pozícióba, az oszlopindex helyébe kell kerülni:

```
RawData[, Inc2 := N / Population * 1e5]
```

A kettő valóban ugyanazt eredményezi:

```
identical(RawData$Inc, RawData$Inc2)
```

```
[1] TRUE
```

Ebben van egy újdonság: az összes eddigi példában *új* táblát hoztunk létre (még ha csak ki is írtuk, és nem mentettük el változóba), ez az első eset, ahol *megelevő* táblát módosítunk. Ez nagyon fontos: mint láthatjuk is, nem kell az eredményt belementenünk egy változóba, azért nem, mert az utasítás lefuttatásakor *maga az eredeti tábla módosult!* Ezt szokták az informatikában referencia szerinti módosításnak¹⁸ hívni. (És igen, ezt az indexelés szintaktikájával éri el a `data.table`, még ha elég meglepő is első látásra.)

Kiírás ilyenkor ugyanúgy nincs, mint általában az értékadásos utasításoknál R-ben. Ha szeretnénk az értékadás után rögtön ki is íratni a táblát akkor egy `[]` jelet kell tennünk a parancs után, pl. `RawData[, Inc2 := N / Population * 1e5][]`.

Használhatjuk ezt a megoldást megelevő változó felülírására, nem csak új létrehozására. Például, ha meggondoljuk magunkat, és az incidenciát per millió fő mértékegységben szeretnénk megadni:

```
RawData[, Inc2 := Inc2 * 10]
```

Egyszerre több változót is definiálhatunk (lehet vegyesen új definiálása és régi felülírása, ennek nincs jelentősége), ennek módszere:

```
RawData[, c("logPop", "sqrtPop") := list(log(Population),  
                                           sqrt(Population))]
```

Mivel az értékadás bal oldalán sztring-vektor áll, így könnyen előállítható gépi úton is. A jobb oldalon pedig lista szerepel, így itt is igaz, hogy nem muszáj kézzel felsorolni, bármilyen olyan függvény szerepelhet ott, ami listát ad vissza.

Változó törölhető is ilyen módon:

¹⁸Ez problémát jelenthet akkor, ha egy függvényen belül csinálunk ilyet, hiszen ez azt fogja maga után vonni, hogy a bemenetként átadott adattábla át fog alakulni. Ez esetben a `copy` függvény segíthet: ezzel készíthetünk első lépésben egy másolatot a tábláról, és ha utána azon dolgozunk, akkor az eredeti, bemenetként megkapott tábla nem fog átalakulni.


```
RawData[, Inc2 := NULL]
```

Ha több változót törölnénk:

```
RawData[, c("logPop", "sqrtPop") := NULL]
```

Mi értelme van mindennek? Az első válasz az, hogy bizonyos esetekben gyorsabb¹⁹. A második, hogy mindez kombinálható a `data.table` többi elemével, tehát a sorindexeléssel és a csoportosítással.

Például szeretnénk a „Budapest” kifejezést lecserélni arra, hogy „Főváros” a megye változóban. Ezt megoldhatjuk így:

```
RawData[County == "Budapest", County := "Főváros"]
```

Tehát: ha az értékadást szűréssel kombináljuk, akkor a nem kiválasztott soroknál nem változik az érték. (Ha pedig nem meglevő változót módosítunk, hanem újat hozunk létre, akkor a nem kiválasztott soroknál NA kerül az új változóba.)

Ezt könnyen megoldhattuk volna másképp is, de nézzük egy izgalmasabb példát. Szeretnénk minden nemre, életkorra, megyére és ráktípusra eltárolni, hogy az adott nemből, életkorból, megyéből és ráktípusból mi volt a legkisebb feljegyzett incidencia (a különböző évek közül, tehát). Ezt `data.table` nélkül csak macerásabban tudnánk megtenni, de a `data.table` használatával nagyon egyszerű (és nagyon logikus) a megoldás:

```
RawData[, MinInc := min(Inc), .(County, Sex, Age, ICDCode)]
```

A csoportosító változót kell használnunk, ami teljesen logikus is: képezi a csoportokat nem, életkor, megye és ráktípus szerint (tehát az egyes csoportokban a különböző évek fognak szerepelni), veszi azok körében az `Inc` minimumát, és azt menti el `MinInc` néven – az adott csoport különböző soraihoz mindig ugyanazt az értéket. Íme:

```
RawData[ICDCode == "C18" & Age == 70 & County == "Főváros"]
```

¹⁹Az R a 3.1.0-s verzió előtt minden ilyen változó-értékadási műveletnél deep copy-t csinált az adatbázisról, ami azt jelenti, hogy nem csak a memóriamutatókat frissítette (ez lenne a shallow copy), hanem az egész adatbázist fizikailag átmásolta egy másik memóriaterületre. Ez nagyon gazdaságtalan, pláne, mert értelmetlen is, hiszen egy új változó definiálásától a meglevő tartalom maradhatna ugyanott. Ezt a 3.1.0-s verzióban orvosolták, de az továbbra is megmaradt, hogy nem az egész oszlop kap értéket, csak egy része, akkor deep copy készül. Ezzel szemben a `data.table` minden esetben és minden verzióban shallow copy-t csinál értékadásnál.

	County	Sex	Age	Year	ICDCode	N	Population	Inc	MinInc
	<char>	<char>	<num>	<num>	<char>	<int>	<num>	<num>	<num>
1:	Főváros	Férfi	70	2000	C18	97	30697.5	315.9866	217.0223
2:	Főváros	Férfi	70	2001	C18	111	32326.5	343.3715	217.0223
3:	Főváros	Férfi	70	2002	C18	108	31711.5	340.5705	217.0223
4:	Főváros	Férfi	70	2003	C18	99	30984.0	319.5198	217.0223
5:	Főváros	Férfi	70	2004	C18	100	30205.5	331.0655	217.0223
6:	Főváros	Férfi	70	2005	C18	97	29194.5	332.2544	217.0223
7:	Főváros	Férfi	70	2006	C18	90	28123.5	320.0171	217.0223
8:	Főváros	Férfi	70	2007	C18	96	27422.5	350.0775	217.0223
9:	Főváros	Férfi	70	2008	C18	95	27080.5	350.8059	217.0223
10:	Főváros	Férfi	70	2009	C18	102	26957.0	378.3804	217.0223
11:	Főváros	Férfi	70	2010	C18	80	27335.5	292.6597	217.0223
12:	Főváros	Férfi	70	2011	C18	97	28288.0	342.9016	217.0223
13:	Főváros	Férfi	70	2012	C18	89	30601.5	290.8354	217.0223
14:	Főváros	Férfi	70	2013	C18	118	32451.0	363.6252	217.0223
15:	Főváros	Férfi	70	2014	C18	108	34269.5	315.1490	217.0223
16:	Főváros	Férfi	70	2015	C18	103	35428.0	290.7305	217.0223
17:	Főváros	Férfi	70	2016	C18	107	35831.5	298.6199	217.0223
18:	Főváros	Férfi	70	2017	C18	101	36012.0	280.4621	217.0223
19:	Főváros	Férfi	70	2018	C18	78	35941.0	217.0223	217.0223
20:	Főváros	Nő	70	2000	C18	115	52434.0	219.3233	161.1007
21:	Főváros	Nő	70	2001	C18	99	53138.5	186.3056	161.1007
22:	Főváros	Nő	70	2002	C18	95	51751.0	183.5713	161.1007
23:	Főváros	Nő	70	2003	C18	110	50498.5	217.8283	161.1007
24:	Főváros	Nő	70	2004	C18	92	49073.0	187.4758	161.1007
25:	Főváros	Nő	70	2005	C18	102	47308.5	215.6061	161.1007
26:	Főváros	Nő	70	2006	C18	74	45934.0	161.1007	161.1007
27:	Főváros	Nő	70	2007	C18	95	45165.0	210.3399	161.1007
28:	Főváros	Nő	70	2008	C18	94	44594.5	210.7883	161.1007
29:	Főváros	Nő	70	2009	C18	84	44558.5	188.5162	161.1007
30:	Főváros	Nő	70	2010	C18	81	45258.5	178.9719	161.1007
31:	Főváros	Nő	70	2011	C18	101	46479.0	217.3024	161.1007
32:	Főváros	Nő	70	2012	C18	104	48132.0	216.0725	161.1007
33:	Főváros	Nő	70	2013	C18	134	50451.0	265.6042	161.1007
34:	Főváros	Nő	70	2014	C18	124	52854.0	234.6085	161.1007
35:	Főváros	Nő	70	2015	C18	94	54534.5	172.3680	161.1007
36:	Főváros	Nő	70	2016	C18	120	55317.5	216.9295	161.1007
37:	Főváros	Nő	70	2017	C18	95	55986.5	169.6838	161.1007
38:	Főváros	Nő	70	2018	C18	113	56508.0	199.9717	161.1007
	County	Sex	Age	Year	ICDCode	N	Population	Inc	MinInc