

SageMath bevezetés

Diszkrét modellek alkalmazásai

A SageMath (röviden Sage) egy komputeralgebra rendszer, amely a matematika különböző területein (algebra, analízis, számelmélet stb.) tud számításokat végezni. A nyelv a Python programozási nyelven alapul, azaz annak szintaxisa és legtöbb funkciója elérhető Sage-ben. A Sage ezenfelül számos speciális matematikai programcsomagra épül (NumPy, SymPy, Maxima, GAP, R stb.), ezek erejét egyesíti, közös felület alá rendezi, és számos új funkcióval egészíti ki.

A Sage szabad szoftver, azaz például nem korlátozza a felhasználót a szoftver használatában.¹

A Sage-ről a legtöbb információ a honlapjáról érhető el:

<https://www.sagemath.org/>

1. Sage elérése

A Sage-et alapvetően kétféle módon tudjuk használni: vagy letöltjük és telepítjük egy számítógépre, vagy csatlakozunk egy Sage szerverhez az interneten keresztül.

A Sage-et telepíteni nem teljesen triviális, mert egy rendkívül komplex szoftvercsomagról van szó. De ha ezt egyszer megtesszük, akkor utána egyszerűen és gyorsan használhatjuk, megbízhatóan (például nem kell hozzá internetkapcsolat), és teljes mértékben mi irányítjuk a számításainkat.

Ha egy internetes Sage szervert használunk, akkor megússzuk a telepítést, de szembe kell néznünk az online szolgáltatásokra jellemző problémákkal: folyamatos internetkapcsolat kell hozzá, az adott szervernek mindig elérhetőnek kell lennie, és a programjaink felett nincs hatalmunk (csak remélni tudjuk, hogy a szerver üzemeltetője nem módosítja és nem törli a fájljainkat, és hogy nem mutatja meg illetékteleneknek). (Ne feledjük: ha egy internetes szolgáltatást használunk, akkor valójában csak valaki másnak a számítógépét használjuk.)

1.1. Sage telepítése

Az alábbiakban áttekintjük a legegyszerűbb telepítési módokat. A Sage elsősorban Linux-típusú rendszerekre készült, ezért ott valamivel egyszerűbb a telepítés és a használat, de Windows-on és macOS-en is lehetséges.

Először persze célszerű ellenőrizni, hogy nincs-e telepítve már a Sage, például Linuxon próbáljuk meg futtatni parancssorban a `'sage'` parancsot (idézőjelek nélkül).

¹A szabad szoftverek előnyeiről lásd pl.: <https://www.gnu.org/philosophy/free-sw.html>

1.1.1. Linuxon rendszergazdaként

A Sage legkönnyebben Linuxra telepíthető, feltéve, hogy van hozzá rendszergazdai (`root`) jogosultságunk, és olyan Linux-disztribúciót használunk, amely tartalmazza a Sage-et csomagként (legtöbbször `sagemath` néven). Ekkor a telepítés a disztribúciónál megszokott módon működik, például parancssorban a következő parancsokkal:

- Ubuntu, Debian: `sudo apt install sagemath`
- Arch Linux: `sudo pacman -S sagemath`
- Fedora: `sudo dnf install sagemath`

1.1.2. Felhasználóként Windows-on és Linuxon

Egyébként a legjobb lehetőségünk az marad, ha letöltünk és telepítünk egy kész Sage csomagot. A legújabb verziókhoz Windows-ra és Linuxra ilyen sajnos már nem adnak, de valamivel régebbi verziókhoz igen (céljainknak ezek a verziók is tökéletesen megfelelnek). Ezeket megtalálhatjuk a következő oldalon:

<https://www.sagemath.org/mirrors.html>

Válasszuk ki bármelyik működő szerveret az oldalon. (A *mirror* egy weboldal másolatát jelenti, amely azért jött létre, hogy enyhítse az eredeti szerver terhelését, ezáltal gyorsítsa a letöltést.) Az **Archived (Outdated)** cím alatt válasszuk ki az operációs rendszert (pl. Windows vagy Linux), és töltsük le a megfelelő (pl. a legújabb elérhető) csomagot.

Windows esetén:

- Töltsük le a telepítőt (ami egy `.exe` fájl).
- Futtassuk, és kövessük az utasításait. (Megjegyzés: a dokumentációt nem szükséges telepíteni, elég az alapsomagot. Megjegyzés 2: a telepítőben jóvá kell hagyni a licenszt, ami értelmetlen dolog, hiszen a Sage szabad szoftver, vagyis a licenz úgysem korlátozza a használatát, akár elfogadjuk, akár nem.)
- Indítsuk el az asztalon megjelenő SageMath ikont (ha több is van, akkor a legegyszerűbb nevű indítja el a parancssort, azaz amelyiknek a nevében nincs "Shell" vagy "Notebook").

Linux esetén:

- Válasszuk ki a CPU architektúráját (valószínűleg *64bit*).
- Töltsük le a megfelelő tömörített fájlt. (Hivatalosan ezek Debianra és Ubuntura vannak, de más Linux-disztribúciókon is működhetnek, esetleg némi plusz munka árán. Hogy milyen disztribúciónk van, azt ezzel a paranccsal ellenőrizhetjük: `cat /etc/*-release`.)
- Csomagoljuk ki a fájlt, pl. a következő paranccsal: `tar -xf <fájlnév>.tar.bz2` (olyan helyre, ahol van elég tárterület – kicsomagolva kb. 12GB lesz –, és van hozzá jogosultságunk).
- Indítsuk el a `sage` nevű futtatható fájlt a kicsomagolt SageMath könyvtárban. Például: `cd <könyvtárnev>/SageMath`, majd `./sage`.

Első futtatáskor eltarthat egy darabig, mire a Sage elindul és használható lesz. (És ha régebbi verziót telepítettünk, első futtatáskor megjelenhetnek bizonyos hibaüzenetek, pl. `DeprecationWarning`, de ezeket nyugodtan figyelmen kívül hagyhatjuk.)

1.1.3. macOS telepítés

A következő oldalon találhatunk Sage telepítőket macOS-re:

https://github.com/3-manifolds/Sage_macOS/releases

- Töltsük le azt a telepítőt (egy `.dmg` fájlt), amelyik megfelel a Mac CPU-típusának (a 2020 utáni Apple Silicon = `arm64`, a korábbi Intel = `x86_64`).
- Nyissuk meg a fájlt, és a megjelenő telepítőablakban húzzuk át a SageMath ikont a jelzett módon az Applications mappába.
- Ha a telepítés befejeződött, a SageMath ikonja megtalálható az alkalmazások között a szokásos módon. (Első futtatáskor lehet, hogy jóvá kell hagyni bizonyos engedélyeket.)
- Ha ezenfelül szeretnénk, hogy a Sage-et a macOS parancssorából is el lehessen indítani a `'sage'` paranccsal, akkor telepítsük a "recommended" csomagot ugyanazon telepítőablakból (ehhez szükség lehet admin jogosultságokra). (Megjegyzés: elég csak a "Sage Script"-et kiválasztani a "Customize" ablakban.)

1.1.4. Egyéb módok

A Sage telepítésének további módjai megtalálhatóak a hivatalos angol nyelvű leírásban:

<https://doc.sagemath.org/html/en/installation/index.html>

1.2. Sage az interneten

1.2.1. SageMathCell

Egy nagyon egyszerű Sage szerver található a következő címen:

<https://sagecell.sagemath.org/>

Csak írjuk be a Sage parancsot, futtassuk Shift+Enter-rel (vagy az Evaluate gombbal), és megjelenik az eredmény. Elmenteni a kódot csak úgy lehet, ha kimásoljuk a számítógépünkön egy szövegszerkesztőbe, és onnan elmentjük (szokásos kiterjesztés: `.sage`).

Vigyázat: gyakran másoljuk ki és mentsük el a kódot, mert ha bezárjuk a böngészőablakot, akkor a munkánk visszavonhatatlanul elvész!

1.2.2. CoCalc

A CoCalc egy sokféle funkcióval rendelkező összetett webes fejlesztőfelület SageMath-hoz (és egyéb szoftverekhez):

<https://cocalc.com/>

Használatához regisztrálni kell. Az ingyenes verzióban bizonyos korlátozások vannak.

2. SageMath alapok

A Sage szintaktikája nagyon hasonló a Pythonéhoz, de nem 100%-ban azonos. A Pythonról a jegyzet második felében lesz szó, itt a Sage alapjairól és matematikai képességeiről lesz szó.

Bővebb leírás a SageMath hivatalos dokumentációjában található: <https://doc.sagemath.org/>

2.1. Sage parancssor

A Sage parancssor egyenként várja a parancsokat, és rögtön visszaadja az eredményt, például:

```
sage: 1+1
2
sage: 2^10
1024
sage: factorial(50)
30414093201713378043612608166064768844377641568960512000000000000
```

Itt a 'sage:' felirat a *prompt*, azaz azután írjuk be a parancsot, és az Enter lenyomásával futtatjuk.

Már a fentiekből is látható, hogy a Sage-nek nincs problémája a nagy számokkal, például a fenti faktoriálist (50!) pontosan kiszámolta mind a 65 számjegyre.

A kalap-jel (^) Sage-ben a hatványozást jelenti. (Megjegyzés: ebben eltér a Pythontól, ahol a hatványozást két csillaggal (**) írjuk, pl. Pythonban $2**3 == 8$, míg Sage-ben $2^3 == 8$.)

Változókat is létrehozhatunk:

```
sage: a = 2+2*4
sage: a
10
sage: (a^10 - 1) / (a - 1)
1111111111
```

Néhány hasznos parancssori funkció:

- Kilépés: 'quit', 'exit' vagy Ctrl+D.
- A "fel" kurzorbillentyű előhívja a legutóbbi parancsot (többszöri lenyomásra az azelőttieket, a "le" pedig előre felé megy). Az előhívott parancsot módosíthatjuk, és Enterrel újra lefuttathatjuk. Ha beírjuk egy parancs elejét, akkor a "fel" gomb előhívja a legutóbbi ilyen parancsot.
- A Tab billentyű kiegészíti a részben beírt parancsot, vagy megjeleníti a lehetőségek listáját. Például ha beírjuk, hogy 'fa', és megnyomjuk a Tab-ot, akkor megjelenik egy lista, amelyben szerepel a `factorial()` függvény. Vagy például ha az 'a=22' értékadás után a következő sorba beírjuk, hogy 'a.', és megnyomjuk a Tab-ot, akkor megjelenik az összes lehetőség, amit a-val tehetünk (például 'a.binary()' kiírja a számot kettes számrendszerben).
- Kérdőjellel (?) információt kérhetünk Sage függvényekről, pl. 'factorial?' (+ Enter) megjeleníti a `factorial()` függvény leírását. A súgóban kurzorbillentyűkkel vagy a Page Up/Down-nal mozoghatunk, és Q-val léphetünk ki.
- Ha a parancssor színe nehezen olvasható (pl. egyes verziókban kék prompt van fekete háttéren), akkor írjuk be, hogy '%colors linux' (ennek nincs köze a Linuxhoz, mint platform), vagy '%colors nocolor' stb. (a lehetőségekhez lásd: '%colors?' vagy '%colors').

- Aláhúzásjellel (`_`) hivatkozhatunk a legutóbbi eredményre, az azelőttire két aláhúzásjellel (`__`), egészen háromig (`---`). Például:

```
sage: 2^3
8
sage: _ + 2
10
sage: (_ + 1) * __
88
```

2.2. .sage fájl beolvasása

Összetettebb programokat külön fájlba érdemes írni. A Sage programok szokásos kiterjesztése `.sage`. (A fájl elkészítéséhez tetszőleges szövegszerkesztő program használható. Legtöbbjük nem ismeri a Sage-et közvetlenül, ezért célszerű beállítani, hogy Pythonként tekintsen rá.)

Például készítsük el a `new.sage` fájlt a következő tartalommal:

```
# Ciklus erre a négy számra:
for n in [1,2,5,10]:
    print(n, 2^n, factorial(n))
```

A kettőskeresztrel (`#`) kezdődő sorok *kommentek*, azaz azokat a rendszer figyelmen kívül hagyja. Kiíráshoz a `print()` függvényt használjuk (mert a parancssorral ellentétben a fájlban lévő parancsoknál nem ír ki mindent automatikusan). A ciklusokról és a listákról a jegyzet második felében lesz szó.

A programot a `load()` függvénnyel olvashatjuk be:

```
sage: load("könyvtárnev/new.sage")
1 2 1
2 4 2
5 32 120
10 1024 3628800
```

Az elérési utat az aktuális könyvtárhoz képest kell megadni, azaz ahonnan a Sage-et indítottuk. Ha a fájl itt van, akkor elég annyit írni, hogy `'load("new.sage")'`. Az aktuális könyvtárat lekérdezhettük vagy módosíthatjuk az alábbi Sage parancsokkal (hasonlóan a megfelelő Linux parancsokhoz):

- `'pwd'`: aktuális könyvtár kiírása (*print working directory*);
- `'cd "/uj/konyvtar"'`: aktuális könyvtár módosítása (*change directory*);
- `'ls'`: fájlok listázása az aktuális könyvtárban.

De Windows-on a könyvtárnevek kissé bonyolultabbak (mert a Sage-et Linuxra írták):

- Ha az `.exe` telepítőt használtuk (a jegyzet elején leírt módon), akkor a `c:`, `d:` stb. meghajtók a `/cygdrive/c`, `/cygdrive/d` stb. könyvtárnevek alatt lesznek, és a `\`-jelek helyett `/`-jeleket kell írni. Például a `d:\path\new.sage`-hez `load("/cygdrive/d/path/new.sage")`-et kell írunk. (Ha szeretnénk átállítani, hogy melyik könyvtárból indul a Sage, akkor nyissuk meg a másik, "SageMath ... Shell" nevű ikont, és írjuk be, hogy `'sage-sethome 'd:\kezdokonyvtar\''`.)
- Ha WSL-t használunk (Windows Subsystem for Linux), akkor a `c:`, `d:` stb. meghajtók a `/mnt/c`, `/mnt/d` stb. nevek alatt lesznek (pl. `d:\path\new.sage` $\rightarrow$ `load("/mnt/d/path/new.sage")`).

A `load()` függvény nem felejt el az előző állapotot (a változók értékeit stb.), hanem azok ismeretében hajtja végre még egyszer a programot. Ez általában nem jelent problémát, mert a program – remélhetőleg – minden változót és függvényt definiál, amelyre szüksége van, ezért azok úgyis felülírják az előző állapotot. Ha azonban mégis problémánk lenne ebből (mert például egy változóval véletlenül felülírtunk egy beépített függvényt, pl. `var()` vagy `sum()`), akkor a `'reset()'` függvénnyel visszaállíthatjuk az eredeti állapotot a következő `'load()'` előtt.

2.3. Szimbolikus számítás

A Sage alapértelmezés szerint *szimbolikusan* számol. Mit jelent ez? Például:

```
sage: 2/3
2/3
```

Ez eddig nem tűnik túl hasznosnak: csak visszaadta, amit beírtunk. De nem azért, mert nem tud számolni. Próbáljuk tovább:

```
sage: 666/999
2/3
sage: 6/3
2
sage: 2/3 + 1/2
7/6
```

Vagyis kiszámolt mindent, és a legegyszerűbb alakban adta vissza.

De miért nem adta a $2/3$ -ra azt, amit a közönséges számológépek szoktak adni (és amit a Python is ad): 0.6666666666666666 ? Mert ez nem a $2/3$ pontos értéke. A pontos érték $2/3$, amit ennél egyszerűbben nem lehet kifejezni. A pontos érték tizedestört alakjában végtelen sok 6-os szerepel, amit semmilyen számológép nem tud mind megjeleníteni. Ezért a közönséges számológépek közelítő értékekkel szoktak számolni. Ezt hívjuk *numerikus* számításnak, ellentétben a szimbolikus (vagy *egzakt*) számítással, amit a Sage alapértelmezésként végez. A numerikus számítás hátránya, hogy a kerekítési hibák összeadódhatnak, és a végeredmény akár jelentősen eltérhet a valódi értéktől.

További példák szimbolikus számításra Sage-ben ($\text{sqrt}(x) = \sqrt{x}$):

```
sage: sqrt(2)
sqrt(2)
sage: sqrt(16)
4
sage: sqrt(8)
2*sqrt(2)
sage: sqrt(2)*sqrt(8)
4
sage: sin(pi/3)
1/2*sqrt(3)
sage: log(e^3)
3
```

De ha szeretnénk, akkor a Sage is tud numerikusan számolni, még hozzá tetszőleges pontossáig:

```
sage: N(sqrt(2))
1.41421356237310
sage: N(sqrt(2), digits=80)
1.4142135623730950488016887242096980785696718753769480731766797379907324784621070
```

Akkor is numerikus lesz az eredmény, ha eleve lebegőpontos számot adtunk meg:

```
sage: sqrt(2.)      # a tizedespont (.) miatt numerikusan számol
1.41421356237310
```

De a numerikus számítás csak közelítő értéket ad:

```
sage: a = sqrt(2)
sage: b = N(sqrt(2))
sage: a^2 - 2
0                                # pontos érték
sage: b^2 - 2
4.44089209850063e-16           # közelítés
```

A szimbolikus számítás azonban többet jelent a számokkal való egzakt számolásnál. A Sage betűkkel is tud szimbolikusán számolni, például:

```
sage: (x+1)^2
(x + 1)^2
```

Megint nem csinált semmit, de most azért nem, mert nem mondtuk meg, hogy mit szeretnénk:

```
sage: expand((x+1)^2)      # zárójelek felbontása
x^2 + 2*x + 1
sage: factor(x^4 - 1)     # szorzattá alakítás
(x^2 + 1)*(x + 1)*(x - 1)
```

Az x egy előre definiált *szimbolikus változó*. Más szimbolikus változókat a `var()` függvénnyel tudunk definiálni:

```
sage: var("a,b")
(a, b)
sage: expand((a+b)^4)
a^4 + 4*a^3*b + 6*a^2*b^2 + 4*a*b^3 + b^4
```

Fontos, hogy ne keverjük össze a szimbolikus változókat a közönséges változókkal: az utóbbiaknak ugyanis konkrét értékük van (pl. 5), míg a szimbolikus változók értéke maga a szimbólum (pl. x):

```
sage: n = 5          # n egy változó
sage: n^2
25
sage: var("n");      # n egy szimbolikus változó
sage: n^2
n^2
```

Szintén ne keverjük össze a szimbolikus változókat az ún. *szimbolikus konstansokkal*: pi, e, i, golden_ratio stb., ezek ugyanis konkrét matematikai értékeket jelentenek:

```
sage: x/4
1/4*x
sage: pi/4
1/4*pi      # eddig látszólag nincs különbség
sage: tan(x/4)
tan(1/4*x)
sage: tan(pi/4)
1           # de most már van
sage: N(e)   # a szimbolikus konstansoknak van numerikus értéke
2.71828182845905
sage: N(x)   # a szimbolikus változóknak nincs
[...]
TypeError: cannot evaluate symbolic expression numerically
sage: i^2
-1
sage: e^(i*pi) + 1  # Euler-összefüggés
0
sage: N(golden_ratio)
1.61803398874989
sage: bool(golden_ratio == (sqrt(5) + 1)/2)
True
```

2.4. Matek a Sage-ben

Sage-ben rendelkezésre állnak az alábbi gyakran használt matematikai függvények:

abs(x)	$ x $
conjugate(x)	$\bar{x}$
factorial(n)	$n!$
binomial(n,k)	$\binom{n}{k}$
sqrt(x)	$\sqrt{x}$
sin(x)	$\sin x$
cos(x)	$\cos x$
tan(x)	$\operatorname{tg} x$
log(x,a)	$\log_a x$
log(x)	$\ln x = \log_e x$
exp(x)	e^x
floor(x)	$\lfloor x \rfloor$
ceil(x)	$\lceil x \rceil$

```
sage: sqrt(-4)
2*I
sage: conjugate(2 + 3*i)
-3*I + 2
sage: binomial(5, 2)
10
sage: var("n"); binomial(n, 2)
1/2*(n - 1)*n
sage: exp(2*log(x))
x^2
sage: floor(57/10)
5
sage: ceil(57/10)
6
sage: (cos(x)^2 + sin(x)^2).simplify_full()
1
```

Megjegyzés: Sage-ben a matematikai függvények többségét nemcsak önálló függvényként írhatjuk (pl. `sqrt(4)`), hanem tagfüggvényként is, pl. `4.sqrt()`, `((x + 1)^2).expand()` stb.

Sage-ben összegezni a `sum()` függvénnyel lehet, amit kétféleképpen is lehet használni. Az egyszerűbb (Pythontól örökölt) változatban csak meg kell adni egy listát (vagy egyéb sorozatot):

```
sage: sum([10,20,30])
60
```

A másik változat a matematikai szummához ($\sum$) hasonlóan használható. Ehhez négy paramétert kell megadni: `sum(<képlet>, <változó>, <mettől>, <meddig>)`. Hasonlóan működik a produktum ($\prod$) is: `product(...)`. Például:

<code>sage: var("n,k");</code>	
<code>sage: sum(k^2, k, 1, 10)</code>	$\sum_{k=1}^{10} k^2 = 1^2 + 2^2 + \dots + 10^2 = 385$
<code>385</code>	
<code>sage: sum(k, k, 1, n).factor()</code>	$\sum_{k=1}^n k = 1 + 2 + \dots + n = \frac{n(n+1)}{2}$
<code>1/2*(n + 1)*n</code>	
<code>sage: # végtelenig (oo == Infinity):</code>	
<code>sage: sum(1/2^n, n, 0, oo)</code>	$\sum_{n=0}^{\infty} \frac{1}{2^n} = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots = 2$
<code>2</code>	
<code>sage: # szorzat ("produktum"):</code>	
<code>sage: product(k^2, k, 1, n)</code>	$\prod_{k=1}^n k^2 = 1^2 \cdot 2^2 \cdot 3^2 \cdot \dots \cdot n^2 = (n!)^2$
<code>factorial(n)^2</code>	

A Sage matematikai képességeinek fontos része az egyenletek (és egyenletrendszerek) megoldása, amelyre a `solve()` függvényt használhatjuk:

```
sage: solve(x^3 + 3*x^2 == 2*x + 2, x)
[x == -sqrt(2) - 2, x == sqrt(2) - 2, x == 1]
sage: _[0].rhs()    # a fentiből az első megoldás kiválasztása
-sqrt(2) - 2
sage: var("x,y");
sage: solve([x + y == 6, x - y == 4], x, y)
[[x == 5, y == 1]]
sage: # Amit nem adunk meg ismeretlennek, az paraméter:
sage: solve(2*x + y^2 == 6, x)
[x == -1/2*y^2 + 3]
sage: solve(2*x + y^2 == 6, y)
[y == -sqrt(-2*x + 6), y == sqrt(-2*x + 6)]
sage: # Ha végtelen sok megoldás van, akkor azokat paraméteresen adja meg:
sage: solve(x^2 + y^2 == 5, x, y)
[[x == r1, y == sqrt(-r1^2 + 5)], [x == r2, y == -sqrt(-r2^2 + 5)]]
sage: # De ha leszűkítjük egész számokra, akkor már csak véges sok megoldás lesz:
sage: assume(x, y, "integer")    # lehet még: "real", assume(x > 0) stb.
sage: solve(x^2 + y^2 == 5, x, y)
[(2, -1), (1, 2), (-1, -2), (2, 1), (-2, -1), (-2, 1), (1, -2), (-1, 2)]
```

A Sage analízisben is erős (határérték, differenciál- és integrálszámítás):

```
sage: var("n");
sage: limit(2*n/(3*n + 1), n = oo)       $\lim_{n \rightarrow \infty} \frac{2n}{3n + 1} = \frac{2}{3}$ 
2/3
sage: limit(1/(x-1), x = 1, dir='-')     $\lim_{x \rightarrow 1^-} \frac{1}{x - 1} = -\infty$ 
-Infinity
sage: diff(x^3 + sin(2*x), x)             $(x^3 + \sin(2x))' = 3x^2 + 2\cos(2x)$ 
3*x^2 + 2*cos(2*x)
sage: diff(x^3 + sin(2*x), x, 2)         $(x^3 + \sin(2x))'' = 6x - 4\cos(2x)$ 
6*x - 4*sin(2*x)
sage: integral(3*x^2 + 2*cos(2*x), x)     $\int (3x^2 + 2\cos(2x)) dx = x^3 + \sin(2x) + C$ 
x^3 + sin(2*x)
sage: integral(sin(x), x, 0, pi)         $\int_0^\pi \sin x dx = 2$ 
2
```

Vektorok létrehozásához használjuk a `vector()` függvényt:

```
sage: v = vector([1,3,-2])
sage: v
(1, 3, -2)
sage: v[2]          # harmadik komponens (0-tól indexel)
-2
sage: w = vector([2,1,0])
sage: v + 2*w
(5, 5, -2)
sage: v*w           # skaláris szorzat
5
sage: v.norm(2)     # v hossza (euklideszi normája)
sqrt(14)
```

Mátrixokat a `matrix()` függvénnyel lehet létrehozni:

```
sage: M = matrix([[3,1,-4], [2,5,6], [1,4,8]])
sage: M
[ 3  1 -4]
[ 2  5  6]
[ 1  4  8]
sage: M[1][2]      # a második sor harmadik eleme
6
sage: M.transpose() # transzponált
[ 3  2  1]
[ 1  5  4]
[-4  6  8]
sage: M.det()      # determináns
26
```

```

sage: M.inverse()
[ 8/13 -12/13      1]
[ -5/13  14/13     -1]
[ 3/26 -11/26     1/2]
sage: M.inverse() * M == matrix.identity(3)
True
sage: matrix([[1,-1,-1], [1,1,0], [3,0,1]]).eigenvalues() # sajátértékek
[1, 1 - 2*I, 1 + 2*I]

```

És mindez szimbolikus változókkal is működik, például így lehet egy általános 2×2 -es mátrixszal paraméteresen számolni:

```

sage: var("a,b,c,d");
sage: M = matrix([[a,b], [c,d]])
sage: M
[a b]
[c d]
sage: M.det()
-b*c + a*d
sage: M^2
[a^2 + b*c a*b + b*d]
[a*c + c*d b*c + d^2]
sage: (M.inverse() * M).simplify_full()
[1 0]
[0 1]
sage: var("x,y");
sage: v = vector([x,y])
sage: M * v
(a*x + b*y, c*x + d*y)
sage: v.norm(2)
sqrt(abs(x)^2 + abs(y)^2)

```

Lineáris egyenletrendszereket többféleképpen is megoldhatunk:

```

sage: A = matrix([[0,1,-3], [4,5,-2], [2,3,-1]])
sage: b = vector([-5,10,7])
sage: A.solve_right(b) # A*x == b ("right": x jobbról szoroz)
(-1, 4, 3)
sage: A*_ == b
True
sage: A.inverse() * b # kevésbé hatékony
(-1, 4, 3)
sage: var("x,y,z");
sage: solve([y - 3*z == -5, 4*x + 5*y - 2*z == 10, 2*x + 3*y - z == 7], x,y,z)
[[x == -1, y == 4, z == 3]]

```

3. Python programozás

Most áttekintjük a Sage programozási alapjait, amelyet a Pythontól örökölt. Ezekről részletesebb információ a Python dokumentációjában található: <https://docs.python.org/>.

3.1. Kiírás

Fent már láttuk, hogy kiíráshoz a `print()` függvényt használjuk:

```
print("Jó reggelt!")
```

Mint látjuk, Pythonban az utasítások végére nem kell pontosvesszőt (;) írni (bár nem is tilos). Viszont ha egy sorba több utasítást írunk, akkor azokat pontosvesszővel kell elválasztani:

```
print(a); print(a*2)
```

Számokat (vagy egyéb objektumokat) többféleképpen is kiírhatunk:

```
print("a =", a)
print("a = " + str(a))    # miért nem jó str() nélkül?
print("a = %d" % a)       # mint a printf() a C-ben
print("a = %d, a^2 = %d" % (a, a^2))
print(f"a = {a}, a^2 = {a^2}")
print("a = {}, a^2 = {}".format(a, a^2))
```

3.2. Típusok

A Python (és így a Sage) egy dinamikusan típusos nyelv, azaz a változók típusát nem kell előre megadni, hanem az értékadásból kiderül, és újabb értékadással megváltoztható.

Az alább áttekintünk néhány fontosabb típust. A számokkal vigyázni kell, mert azok Pythonban és Sage-ben különböző típusúak.

Csak Python:

- `int`: egész szám, pl. 123
- `float`: lebegőpontos szám, pl. 3.14

Csak Sage:

- `Integer`: egész szám, pl. 123
- `RealNumber`: lebegőpontos szám, pl. 3.14
- `Rational`: racionális törtszám, pl. 2/3
- `Expression`: szimbolikus kifejezés, pl. $(x + 1)^2$ vagy `sqrt(2)`

Python és Sage:

- `str`: szöveg [*string*], azaz karaktersorozat, pl. "alma", "123" vagy "" (üres szöveg). Szövegeket összefűzhetünk a plusz jellel (+), pl. `s = "alma"; print(s + "fa")`.
- `list`: elemek listája, pl. [3, 1, 4, 1, 5] (a listákról később részletesen lesz szó).
- `bool`: logikai érték, azaz igaz (`True`) vagy hamis (`False`). Az összehasonlító relációk ilyen típust adnak eredményül, pl. `b = 3 > 2` ugyanaz, mint `b = True`. (Lásd az `if`-utasítást később.)
- `NoneType`: egyetlen értéke a `None`, azaz "semmi". Azt jelzi, hogy egy érték hiányzik, például egy függvény nem ad vissza semmit, vagy egy paramétere nincs megadva.

Az értékek típusát a `type()` függvénnyel kérdezhetjük le, például:

```
a = "alma"
print(type(a))          # <class 'str'>
print(type([1,2]) == list) # True
```

Az értékeket konvertálhatjuk más típusúvá, ha a típus nevét függvényként használjuk, például:

```
print(12 == "12")       # False (egyik Integer, másik string)
print(str(12) == "12")  # True
print(12 == Integer("12")) # True
```

3.3. Elágazás (if)

Pythonban az elágazási struktúra teljes formájában így néz ki:

```
if a > 0:
    print("pozitív")
elif a == 0:
    print("nulla")
else:
    print("negatív")
```

Amint látható, Pythonban a program struktúráját nem írásjelek jelzik (mint pl. C-ben { és }), hanem a "behúzás" [*indentation*], azaz a sor elején lévő szóközök vagy tabulátorok. A vezérlési szerkezetet kettőspont (:) választja el a benne lévő utasításoktól.

A vezérlési szerkezet tartalma addig tart, amíg a sorok bejebbek kezdődnek, mint a vezérlési szerkezet kezdete. Például:

```
if a > 2:
    print("Ez akkor fut le, ha 'a' nagyobb mint 2.")
    print("Ez is.")
if a <= 5:
    print("Ez akkor fut le, ha 'a' legfeljebb 5, függetlenül az előző if-től.")
    if a >= 1:
        print("Ez akkor fut le, ha 'a' 1 és 5 között van,")
        print("mert ez az 'if' az előző if-utasításon belül van.")
    print("Kiléptünk a belső if-ből, de a külső még mindig aktív,")
    print("vagyis ez akkor fut le, ha a <= 5.")
print("Ez mindig lefut.")
```

(Vigyázat: ne keverjük össze a tabulátorokat és a szóközöket! Nekünk ugyanúgy nézhetnek ki, de a gép számára nem!)

Az egyszerű if-utasítást egy sorba is írhatjuk:

```
if a == 4: print("négy")
```

Nemcsak if-utasítás, hanem if-kifejezés is van:

```
print("abs(a) =", a if a > 0 else -a)
```

Az 'if' feltételében tetszőleges logikai kifejezés (bool érték) állhat. Ehhez használhatjuk az összehasonlító műveleteket: ==, !=, <, >, <=, >=. Vigyázat: az egyenlőségvizsgálat két egyenlőségjel (if a == 5), az értékadás pedig egy (a = 2), ne keverjük őket össze.

Logikai kifejezéseket összekapcsolhatunk az 'and', 'or' és 'not' logikai műveletekkel, például a következő utasítások mind ekvivalensek:

```
if a >= 2 and a <= 5:    print("2 és 5 között")
if a >= 2 and not a > 5: print("2 és 5 között")
if not (a < 2 or a > 5): print("2 és 5 között")
```

Ha egy változó eleve logikai (bool) típusú, akkor nem kell összehasonlítani a True/False értékkel, hanem közvetlenül használható az if-ben, például:

```
b = a > 10    # b értéke True vagy False
if b: print("OK")          # rövid és tömör
if b == True: print("OK")  # ugyanaz, de feleslegesen hosszú
# Ezt is lehet (csak nem túl hasznos):
if True: print("Mindig lefut.")
if False: print("Sosem fut le.")
```

Sőt, nemcsak bool, hanem tetszőleges típusú értéket írhatunk az if-be összehasonlítás nélkül. Ilyenkor a feltétel azt jelenti, hogy az érték "nem nulla", "nem üres" stb. Például:

```
v = 12 # vagy v = "abc", vagy v = [1,2]
if v:   print("igaz")
if not v: print("hamis")
v = 0   # vagy v = "", vagy v = []
if v:   print("hamis")
if not v: print("igaz")
```

Az 'if v' feltétel tehát – típustól függően – annak a rövidítése, hogy 'if v != 0', 'if v != ""', 'if len(v) != 0' stb., az 'if not v' pedig ezek ellenkezője ('if v == 0' stb.).

3.4. Ciklusok (for, while)

A következő ciklus kiírja a számokat 10-től 19-ig:

```
for i in range(10, 20):
    print(i)
```

Pythonban a tartományok eleje beleszámít a felsorolásba, a vége viszont nem, ezért megy a ciklus csak 19-ig, pedig 20-at adtunk meg végpontnak. (Megjegyzés: ez konzisztens a listák indexelésével, ahol, mint látni fogjuk, egy 10-elemű listát 0-tól 9-ig tudunk indexelni.)

A következő ciklus pedig végigmegy a megadott lista minden elemén, és kiírja azok négyzeteit (9, 1, 16, 1 stb.):

```
for i in [3,1,4,1,5,9]:
    print(i*i)
```

Általában a `for`-ciklus `'in'` szava után bármit írhatunk, ami felsorolható, például listát, szöveget (ami karakterek sorozata) vagy generátort (lásd a sorozatokat később). Ilyen például a fenti `range()`, amely számok számtani sorozatát generálja. Például:

```
for i in range(2, 10): print(i)      # 2 3 4 5 6 7 8 9
for i in range(10): print(i)        # 0 1 2 3 4 5 6 7 8 9
for i in range(0, 10, 2): print(i)  # 0 2 4 6 8
for i in range(10, 0, -1): print(i) # 10 9 8 7 6 5 4 3 2 1
for i in range(10, 0): print(i)     # (nem ír ki semmit, mert 10 >= 0)
```

A ciklusok egymásba ágyazhatóak, például a következő program kiír egy szorzótáblát 1-től 10-ig:

```
for i in range(1, 11):
    for j in range(1, 11):
        print("%2d" % (i*j), end=" ") # end: nincs sortörés
    print() # de a sor végén legyen sortörés
```

Ha a felsorolás elemei összetettek, akkor a ciklusváltozó is lehet hasonlóan összetett:

```
for (x,y) in [(1,2),(-2,3),(5,6)]: # vagy: for x,y in ...
    if x > 0:
        print(f"x={x}, y={y}")
```

A `while`-ciklus (*előltesztelő ciklus*) addig fut, amíg a feltétel igaz. Formailag hasonló az `if`-utasításhoz. Például a következő két ciklus ekvivalens:

```
# for-ciklus 0-tól 9-ig:
for i in range(10):
    print(i)
# while-ciklus 0-tól 9-ig:
i = 0
while i < 10:
    print(i)
    i += 1 # más szóval: i = i + 1 (Pythonban nincs 'i++')
```

A ciklusokban használhatók a más nyelvekben megszokott `break` és `continue` utasítások: a `break` megszakítja a ciklust, és az azutáni utasításra ugrik, a `continue` pedig csak a ciklus aktuális iterációját szakítja meg, és a következő iterációt (pl. `'i'`-értéket) kezdi el.

Megjegyzés: Pythonban nincs *hátultesztelő ciklus*, azaz a C-ben megszokott `do-while`-ciklus.

3.5. Függvények

Pythonban a következőképpen lehet függvényt definiálni:

```
def osszead(a, b):
    return a + b
```

Ezt aztán így lehet meghívni:

```
print(osszead(2, 3))
```

Ez persze még nem egy túl érdekes függvény. Nézzünk valami bonyolultabbat:

```
def szamjegyek(n, base=10):
    dig = []
    while n > 0:
        dig.append(n % base)    # maradékos osztás maradéka
        n = n // base          # maradékos osztás hányadosa
    dig.reverse()
    return dig
print(szamjegyek(127))          # [1, 2, 7]
print(szamjegyek(127, 2))      # [1, 1, 1, 1, 1, 1, 1]
print(szamjegyek(n=127, base=4)) # [1, 3, 3, 3]
```

Megjegyzés: Pythonban egy függvény (vagy egyéb vezérlési szerkezet, pl. 'if') törzse formailag nem lehet üres. Ezért ha mégis szeretnénk üresen hagyni (mert pl. később fogjuk csak megírni), akkor használjuk a `pass` utasítást, amely nem csinál semmit:

```
def valami(n):
    pass
valami(2)    # nem csinál semmit, de nem is okoz hibát
```

Hasonlóan:

```
if 3 == 3:
    pass
else:
    print("lehetetlen")
```

3.6. Listák

A Python egyik legfontosabb összetett típusa a *lista* (`list`). Egy listát legegyszerűbben az elemei felsorolásával adhatunk meg, szögletes zárójelek között:

```
[3, 1, 4, 1, 5]
["alma", "körte"]
[]                # üres lista
[123, [1,2,3], "alma"] # inhomogén lista (különböző típusú elemekkel)
```

Listát konvertálhatunk másfajta sorozatból a `list()` függvénnyel:

```
print(list(range(5)))    # [0, 1, 2, 3, 4]
print(list("alma"))      # ["a", "l", "m", "a"]
```

Listát létrehozhatunk képlettel is a `for-in` konstrukcióval, a matematikai halmazjelöléshez hasonló módon (pl. $\{x^2 \mid x \in \{0, \dots, 4\}\}$):

```
print([i*i for i in range(5)]) # négyzetszámok 0-tól 16-ig: [0, 1, 4, 9, 16]
print([i*i for i in range(10) if i%2 != 0]) # páratlan négyzetszámok 1-től 81-ig
print([2*i for i in [3,1,4,1,5]]) # [6,2,8,2,10]
print([i*j for i in range(1,6) for j in range(1,6)]) # szorzótábla listákkal
```

Listák elemeit szögletes zárójellel lehet lekérdezni vagy módosítani (az indexelés 0-tól indul):

```
L = [3,1,4,1,5,9]
print(L[0])    # 3
print(L[1])    # 1
print(L[2])    # 4
print(L[4])    # 5
print(L[6])    # IndexError: list index out of range
print(L[-1])   # 9    (visszafelé is lehet indexelni)
print(L[-2])   # 5
print(L[-3])   # 1
L[2] = 6
print(L)       # [3,1,6,1,5,9]
L[10] = 10     # IndexError: list assignment index out of range
```

Egy lista elemeinek számát a `len()` függvénnyel kérdezhetjük le:

```
print(len([3,1,4,1,5])) # 5
print(len([]))          # 0
print(len([[3,1,4]]))   # 1 (egy eleme van, ami szintén egy lista)
```

A következő két ciklus ekvivalens (mindkettő kiírja a lista elemeit):

```
L = [3,1,4,1,5,9]
for a in L:
    print(a)
for i in range(len(L)):
    print(L[i])
```

Listákhoz lehet elemeket hozzáadni vagy törölni:

```
L = [3,1,4,1,5]
L.append(9)      # 9 beszúrása a végére
print(L)        # [3,1,4,1,5,9]
L.pop()         # utolsó elem (9) törlése
print(L)        # [3,1,4,1,5]
L.pop(2)        # L[2] (== 4) törlése
print(L)        # [3,1,1,5]
L.insert(1, 9)   # 9 beszúrása második elemnek (L[1])
print(L)        # [3,9,1,1,5]
```

Listákat lehet összeadni és számmal szorozni:

```
L1 = [3,1]
L2 = [4,1,5]
print(L1 + L2)  # [3,1,4,1,5]
print(L2 + L1)  # [4,1,5,3,1]
print(3 * L1)   # [3,1,3,1,3,1]
print([2] * 5)  # [2,2,2,2,2]
```

Listákat lehet *szeletelni*, azaz egyszerre több elemüket kivenni:

```
L = [3,1,4,1,5,9]
print(L[3:5])    # [1,5]    (L[3]-tól L[5] előttig, azt már nem beleértve)
print(L[3:])     # [1,5,9]  (L[3]-tól kezdve az összes elem)
print(L[:3])     # [3,1,4]  (összes elem L[3] előttig)
print(L[:])      # [3,1,4,1,5,9] (összes elem, azaz L másolata, lásd lent)
print(L[3::2])   # [1,9]    (L[3]-tól kezdve minden második elem)
print(L[::-1])   # [9,5,1,4,1,3] (az összes elem visszafelé)
L[-4:-1] = [2,2] # L[-4]-tól 3 elemet lecserélünk [2,2]-re
print(L)         # [3,1,2,2,9]
```

Ha Pythonban egy listát (vagy egyéb összetett objektumot) értékül adunk egy másik változónak, akkor abból nem készül másolat, hanem az új változó is ugyanarra az objektumra fog hivatkozni. Ezért lehet szükség a `L[:]` konstrukcióra, amely tényleg lemásolja a listát. Például:

```
L1 = [3,1,4]
L2 = L1      # nem másolat, hanem L1 és L2 ugyanarra a listára fog hivatkozni
L2[1] = 2    # mindkettő módosul!
print(L1)    # [3,2,4]
print(L2)    # [3,2,4]
L3 = L1[:]   # ez már tényleg másolat, vagyis két külön lista lesz
L3[0] = 1    # csak L3 módosul
print(L1)    # [3,2,4]
print(L3)    # [1,2,4]
```

3.7. Egyéb sorozatok

A listához nagyon hasonló a **tuple**, azaz a "rendezett n-es" (pár, hármas, négyes stb.). A fő különbség az, hogy a **tuple** létrehozás után már nem módosítható. Egyébként ugyanazt lehet vele csinálni, csak kerek zárójelekkel kell létrehozni:

```
T = (3,1,4)
print(T[-1])    # 4
print(T[1:])    # (1,4)
print(T + (1,5)) # (3,1,4,1,5)
T[2] = 6        # TypeError: 'tuple' object does not support item assignment
T = (3,)        # egyelemű tuple (vessző nélkül mit kapnánk?)
```

A szöveg (**str**) is sorozat: karakterek sorozata. Idézőjelek között lehet megadni, többféleképpen is (az első két sor ekvivalens):

```
print("I'm OK.")
print('I\'m OK.')
print("""többsoros
szöveget
így lehet megadni""")
```

Hasonló műveletei vannak, mint a listáknak, csak a `tuple`-höz hasonlóan a szöveg sem módosítható, ezenkívül vannak saját műveletei is:

```
S = "almafa"
print(S[2])          # "m"
print(S[1:-1])       # "lmaf"
print("a"*10 + "bc") # "aaaaaaaaaabc"
print(S.replace("a", "A")) # "AlmAfA" (de S maga nem változott)
print("abc,d,e,,fg".split(",")) # ["abc", "d", "e", "", "fg"]
print(",".join(["a","bc","def"])) # "a,bc,def"
```

A *generátor* egy olyan absztrakt sorozat, amely egymás után állítja elő az elemeket, de azokat nem tárolja el egyszerre a memóriában (ellentétben pl. egy listával). Ilyen generátor például a `for`-ciklusoknál megismert `range()`. Például a `range(1000000)` nem tárolja el a számokat 0-tól 999999-ig, csak egyenként adagolja nekünk. (Ha valami okból mégis szeretnénk eltárolni, akkor konvertáljuk listává: `list(range(1000000))`.)

Generátort hasonlóan lehet definiálni, mint egy függvényt, csak nem a `'return'` utasítást használjuk (amely megszakítja a függvény futását), hanem a `'yield'`-et, amely egyenként adagolja az értékeket. Például a `range(a,b)`-t így tudjuk újraírni:

```
def myrange(a, b):
    i = a
    while i < b:
        yield i
        i += 1
```

És ez ugyanúgy használható, mint a `range()`:

```
for i in myrange(2, 10):
    print(i*i)
```

Mivel a generátor nem tárolja el egyszerre az elemeket, ezért akár végtelen generátort is írhatunk:

```
def negyzetszamok():
    i = 1
    while True: # végtelen ciklus
        yield i*i
        i += 1
```

Ezt persze csak úgy használhatjuk, ha a generátort használó ciklus magától megáll:

```
for x in negyzetszamok():
    if x > 1000:
        break
    print(x)
```

3.8. Osztályok

Az osztály [*class*] egy felhasználó által definiált típus, amelyben definiálhatunk adattagokat [*field*] és tagfüggvényeket [*method*]. Például:

```
class Person:
    def __init__(self, name, age): # konstruktor (self = az objektum maga)
        self.name = name # adattagok
        self.age = age
    def greet(self, othername): # tagfüggvény
        print("Üdv, "+othername+"! "+self.name+" vagyok, "+str(self.age)+" éves.")
p = Person("Peti", 36) # konstruktor meghívása (self: p, name: "Peti", age: 36)
p.greet("Tomi") # greet() meghívása (self: p, othername: "Tomi")
```

A tagfüggvények első paramétere (*self*) maga az objektum, amelyen meghívtuk a tagfüggvényt, ezért azt nem kell meghíváskor külön megadni. A konstruktor (*__init__()*) egy speciális tagfüggvény, amely az objektum létrehozásakor fut le.

Örökléssel [*inheritance*] egy létező osztályból újat hozhatunk létre:

```
class NicePerson(Person): # örökli Person minden tulajdonságát
    def greet(self, othername): # greet() felülírása
        Person.greet(self, "kedves "+othername)
j = NicePerson("Juli", 25) # Person örökölt konstruktora (és adattagjai)
j.greet("Timi") # NicePerson felülírt greet() tagfüggvénye
```

Az ún. speciális tagfüggvények neve dupla aláhúzásjellel kezdődik és végződik, és segítségükkel különféle funkciókat (például matematikai műveleteket) definiálhatunk. Például:

```
class Vector:
    def __init__(self, x, y): # konstruktor
        self.x = x
        self.y = y
    def __add__(a, b): # a + b
        return Vector(a.x + b.x, a.y + b.y)
    def __str__(v): # konvertálás szöveggé (pl. print()-hez)
        return "(" + str(v.x) + ", " + str(v.y) + ")"
u = Vector(2, 3)
v = Vector(3, -1)
print(u + v) # (5, 2)
```

Néhány példa speciális tagfüggvényre:

<code>__add__(a, b)</code>	<code>a + b</code>	<code>__pow__(a, b)</code>	<code>a ^ b</code>	<code>__eq__(a, b)</code>	<code>a == b</code>
<code>__sub__(a, b)</code>	<code>a - b</code>	<code>__neg__(a)</code>	<code>-a</code>	<code>__ne__(a, b)</code>	<code>a != b</code>
<code>__mul__(a, b)</code>	<code>a * b</code>	<code>__pos__(a)</code>	<code>+a</code>	<code>__lt__(a, b)</code>	<code>a < b</code>
<code>__truediv__(a, b)</code>	<code>a / b</code>	<code>__abs__(a)</code>	<code>abs(a)</code>	<code>__gt__(a, b)</code>	<code>a > b</code>
<code>__floordiv__(a, b)</code>	<code>a // b</code>	<code>__len__(a)</code>	<code>len(a)</code>	<code>__le__(a, b)</code>	<code>a <= b</code>
<code>__mod__(a, b)</code>	<code>a % b</code>	<code>__bool__(a)</code>	<code>if a:</code>	<code>__ge__(a, b)</code>	<code>a >= b</code>

3.9. Tesztelés

Ahhoz, hogy meggyőződhessünk róla, hogy a megírt programunk helyes, *tesztelnünk* kell azt. Például tegyük fel, hogy két szám szorzatát ismételt összeadással szeretnénk kiszámítani, és ehhez megírtuk a következő – lassú és hibás – függvényt:

```
def mul (a, b):  
    c = a  
    for i in range(1, b):  
        c += a  
    return c
```

Ha ezt egy példára lefuttatjuk, akkor úgy tűnik, hogy működik:

```
sage: mul(24, 100)  
2400
```

De ne elégedjünk meg ennyivel! Teszteljük a függvényt az összes lehetséges számra 0-tól 100-ig:

```
for a in range(101):    # ne feledjük: range(n) csak n-1-ig megy!  
    for b in range(101):  
        assert mul(a,b) == a*b
```

Az ‘assert’ ellenőrzi, hogy a megadott feltétel igaz-e, és ha nem, akkor hibát ad, és megszakítja a programot. (Ugyanezt meg lehetne oldani egy ‘if’-fel, egy `print()`-tel és néhány ‘break’-kel is, de az ‘assert’ sokkal egyszerűbb.)

Ha ezt lefuttatjuk, akkor egy hosszú hibaüzenetet kapunk, amelynek a végén ez áll:

AssertionError:

De ebből még nem derült ki, hogy mely számokra volt hibás. Ehhez egészítsük ki az ‘assert’ sorát:

```
for a in range(101):  
    for b in range(101):  
        assert mul(a,b) == a*b, f"mul({a},{b})"
```

Az ‘assert’ második paraméterében megadhatunk egy szöveget, amelyet hiba esetén ki fog írni. (Itt egy formázott szöveget adtunk meg, amelyben az a és b változók értéke is szerepel, lásd fent, a Python-rész elején a formázott kiírást.) Ekkor a hibaüzenet vége így fog kinézni:

AssertionError: mul(1,0)

És valóban, `mul(1,0) == 1`, nem pedig 0. Vegyük észre, hogy a függvény 0-ra nem működik helyesen. Ezt úgy javíthatjuk ki, ha az összegzést 0-tól indítjuk, és eggyel többször adunk hozzá a-t:

```
def mul (a, b):  
    c = 0  
    for i in range(b):  
        c += a  
    return c
```

Erre már lefut minden teszt.

Ha nagyobb számokra is szeretnénk tesztelni, arra a fenti módszer már túl lassú. Ehelyett (vagy ezenfelül) tesztelhetjük véletlenszerűen is. Véletlen számokat a `randrange()` függvénnyel állíthatunk elő, amely hasonlóan paraméterezhető, mint a `range()`:

```
sage: randrange(100)          # véletlen szám 0 és 99 között
94
sage: randrange(100, 200, 10) # 100-tól 190-ig 10-esével
180
```

(Megjegyzés: Python-ban a `randrange()`-hez szükség van a következő utasításra a program elején: `'from random import randrange'` – Sage-ben erre nincs szükség, mert ott mindig rendelkezésre áll.)

Véletlen számokkal például így tesztelhetjük a függvényt:

```
for i in range(1000):
    a = randrange(10000)
    b = randrange(10000)
    assert mul(a,b) == a*b, f"mul({a},{b})"
```

Ennyi tesztelés után már elég bizonyosak lehetünk benne, hogy a függvény jól működik. De van még egy fontos szempont: milyen gyors? Sage-ben a `'time'` utasítással mérhetjük meg egy parancs futási idejét, például:

```
sage: time mul(1000, 123456789)
CPU times: user 12.9 s, sys: 0 ns, total: 12.9 s
Wall time: 12.9 s
123456789000
sage: time 1000 * 123456789
CPU times: user 16 µs, sys: 0 ns, total: 16 µs
Wall time: 21.5 µs
123456789000
```

Sokmindent írt ki, de figyeljünk csak a "Wall time"-ra (ami a ténylegesen eltelt idő): a `mul()` függvény kb. 13 másodperc alatt futott le, míg a beépített szorzás kb. 21 *milliomod* másodperc alatt – vagyis lényegében azonnal. (Megjegyzés: ms = 0,001 s, µs = 0,000 001 s, ns = 0,000 000 001 s.) A függvényünk tehát – nem meglepő módon – nagyon lassú.

4. Szerző

Ezt a jegyzetet Uray M. János írta, részben felhasználva az alábbi forrásokat:

- [1] <https://compalg.elte.gitlab-pages.hu/dimoa-web/tematika/dimoa>
- [2] <https://doc.sagemath.org/>
- [3] <https://docs.python.org/>
- [4] <https://en.wikipedia.org/wiki/SageMath>

Hibákat, észrevételeket itt lehet jelezni: uray.janos@inf.elte.hu.