SageMath bevezetés

Diszkrét modellek alkalmazásai

A SageMath (röviden Sage) egy komputeralgebra rendszer, amely a matematika különböző területeit (algebra, analízis, számelmélet stb.) fedi le. Szabad szoftver (licensz: GNU GPLv3), azaz pl. nem korlátozza a felhasználót a szoftver használatában.
A Sage nyelve egy Python-alapú programozási nyelv. A Sage sok más speciálisabb matematikai programcsomagra épül (NumPy, SymPy, Maxima, GAP, R stb.), ezeket közös felület alá rendezi, emellett saját funkcióival egészíti ki őket.
A Sage honlapja: https://www.sagemath.org/

1. Sage elérése

A Sage-et alapvetően kétféle módon tudjuk használni: vagy letöltjük és telepítjuk 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 (pl. nem kell hozzá internetkapcsolat), és teljes mértékben kontrollálhatjuk 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 más számítógépét használjuk.)

1.1. Sage telepítése

Az alábbiakban áttekintjük a Sage telepítésének legegyszerűbb módjait.
A többi lehetőség megtalálható a hivatalos angol nyelvű leírásban:
https://doc.sagemath.org/html/en/installation/index.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 telepítéshez csak egy egyszerű parancsot kell kiadnunk, például:

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

1.1.2. Felhasználóként Windows-on és Linux-on

Egyébként szükségünk van egy kész Sage csomagra. A legújabb verziókhoz (jelenleg 10.4) ilyet nem adnak, de a valamivel régebbiekhez (pl. 9.3) igen, Windows-ra és Linux-ra is (többek között). (Céljainknak ezek a verziók is tökéletesek megfelelnek.) Ezeket megtalálhatjuk a következő oldalon: https://www.sagemath.org/mirrors.html
Válasszunk egy szervert 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) csomagot.
Windows esetén:

- Töltsük le a telepítőt (ami egy `.exe` fájl).
- Futtassuk, és kövessük az utasításait. (Megjegyzés: a dokumentációt nem szükséges telepíteni, elég az alapcsomagot. Megjegyzés 2: a telepítőben jóvá kell hagyni a licenszt, ami értelmetlen dolog, hiszen a Sage szabad szoftver, vagyis a licensz nem korlátozza a használatát, akár elfogadjuk, akár nem.)
- Indítsuk el az asztalon megjelenő SageMath ikont.

Linux esetén:

- Válasszuk ki a CPU architektúráját (ami 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. Az egyetemi gépeken jelenleg Debian 11 van, amit ezzel a paranccsal ellenőrizhetünk: '`cat /etc/*-release`'.)
- Csomagoljuk ki a fájlt, pl. a következő paranccsal (konzolban): '`tar -xf <fájlnév>.tar.bz2`'. Megjegyzés: az egyetemi gépeken a saját könyvtár (*home*, azaz `~/`) nem elég nagy ehhez (a csomag kb. 3GB tömörítve, és 12GB kicsomagolva), mert hálózati meghajtón vannak, ahol kvóták vannak. Ezért találnunk kell egy olyan helyet a számítógépen, amelyhez van jogosultságunk (nálam `/vms/` működött).
- Indítsuk el a `sage` nevű futtatható fájlt a kicsomagolt `SageMath` könyvtárban. Például: '`cd <könyvtárnév>/SageMath`', majd '`./sage`'). Először eltarthat egy darabig, mire elindul.

1.2. Sage az interneten

1.2.1. SageMathCell

Egy nagyon egyszerű Sage szerver található a https://sagecell.sagemath.org/ oldalon.
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 egy szövegszerkesztőbe, és elmentjuk a számítógépünkre (`.sage` kiterjesztéssel).

1.2.2. CoCalc

A CoCalc, https://cocalc.com/, egy sokféle funkcióval rendelkező összetett webes fejlesztőfelület SageMath-hoz (és egyéb szoftverekhez).
Használatához regisztrálni kell. Az ingyenes verzióban mindenféle korlátozások vannak.

2. Python alapok

Mivel a Sage egy Python-alapú nyelv, ezért először a Pythonnal kell megismerkedni.
A Python egy általános célú programozási nyelv, amely gyors fejlesztést tesz lehetővé dinamikus típusosságával és rugalmasságával, azonban hatékonyság szempontjából nem a legjobb választás.
A Python honlapja: https://www.python.org/, dokumentációja: https://docs.python.org/.

2.1. Néhány egyszerű parancs

A kommentek kettőskereszttel (#) kezdődnek:

```
# Ez egy komment.
```

A `print()` függvénnyel lehet kiírni valamit:

```
print("Jó reggelt!")
```

A változókat nem kell deklarálni, elég értéket adni nekik:

```
a = 2+3*4
```

Az értékadás nem ír ki semmit, úgyhogy írjuk ki most:

```
print(a)
```

Amint látszik, Pythonban nem kell pontosvesszővel (`;`) lezárni az utasításokat (de nem is tilos). Ha viszont egy sorba írunk több utasítást, akkor azokat pontosvesszővel kell elválasztani:

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

Egy számot többféleképpen is kiírhatunk:

```
print("a =", a)
print("a = " + str(a))   # mi történik str() nélkül?
print("a = %d" % a)      # mint a printf() a C-ben
print(f"a = {a}")
```

Megjegyzés: további információt a `help(`*parancs*`)` utasítással lehet kérni, pl. `help(print)`.

2.2. Típusok

A Python egy dinamikusan típusos nyelv, azaz a változók típusát nem kell előre megadni, hanem az értékadásból kiderül, és újabb értékadással megváltozhatható.
Néhány fontosabb típus:

- `int`: egész szám, pl. `123`
- `str`: szöveg (string), azaz karaktersorozat, pl. `"alma"`, `"123"` vagy `""` (üres szöveg). Szövegeket összefűzhetünk a plusz jellel (+), pl. `s = "alma"; print(s + "fa")`.
- `float`: lebegőpontos szám, pl. `3.14`
- `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.)
- `list`: elemek listája, pl. `[3, 1, 4, 1, 5]` (erről is lesz még szó később).

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

```
a = "alma"
print(type(a))           # <class 'str'>
print(type(12) == int)   # True
```

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

```
a = 12             # type(a) == int
print(a == "12")   # False (egyik int, másik string)
a = str(a)         # type(a) == str
print(a == "12")   # True
```

2.3. 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 bejjebb kezdődnek, mint a vezérlési szerkezet kezdete. Például:

```
if a > 2:
   print("Ez akkor fut le, ha 'a' több 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")
```

A feltételban 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")         # ez a preferált
if b == True: print("OK") # ugyanaz, de feleslegesen hosszú
# Ezt is lehet (csak minek):
if True: print("Mindig lefut.")
if False: print("Sosem fut le.")
```

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

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

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

2.4. Ciklusok (`for`, `while`)

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

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

Python-ban a tartományok elejét beleértjük a felsorolásba, de a végét 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.)

Ez a ciklus pedig a megadott lista elemeinek a négyzeteit írja ki (9, 1, 16 stb.):

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

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=" ")  # nincs sortörés minden szám után
   print()   # de a sor végén van sortörés
```

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

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

A `while`-ciklus addig fut, amíg a feltétel igaz. Például a következő ciklus ekvivalens a `for i in range(10): print(i)` ciklussal:

```
i = 0
while i < 10:
   print(i)
   i += 1    # más szóval: i = i + 1
```

A ciklusokban használhatók a más nyelvekben megszokott `break` és `continue` utasítások.
Megjegyzés: Pythonban nincs "hátultesztelős" ciklus, azaz a C-ben megszokott `do-while`-ciklus.

2.5. 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` parancsot, ami 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")
```

2.6. Listák

A Python egyik legfontosabb összetett típusa a *lista* (`list`).

Egy listát legegyszerűbben az elemei felsorolásával adhatunk meg, szögletes zárójelek között, pl.: `[3,1,4,1,5]` vagy `[]` (üres lista). Pythonban a listák lehetnek inhomogének, azaz az elemeknek nem kell azonos típusúaknak lennie, pl.: `[1, "alma", False, [1,2,3], 123]`.

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. $\{x^2 \mid x \in \{0,\ldots,4\}\}$):

```
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:

```
L = [3,1,4,1,5,9]
print(L[0])   # 3   (a Python 0-tól indexel)
print(L[1])   # 1
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
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 lehet lekérdezni, például a következő két ciklus ekvivalens:

```
L = [3,1,4,1,5,9]
for elem in L: print(elem)
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]
```

(Lásd még: `help(list)`.)

Listákat lehet összeadni és szorozni:

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

Listákat lehet *szeleteni*, azaz egyszerre több elemet 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]-től 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 fog másolat készülni, 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]
```

2.7. Egyéb sorozatok

A listához nagyon hasonló a "rendezett n-es" (`tuple`), csak az 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)
print(2*T)        # (3,1,4,3,1,4)
T[2] = 6          # TypeError: 'tuple' object does not support item assignment
```

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 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 (ami megszakítja a függvény futását), hanem a `yield`-et, ami egyenként adagolja az értékeket. Például ha a `range(a,b)`-t szeretnénk újraírni:
```
def myrange(a, b):
   i = a
   while i < b:
      yield i
      i += 1
```
És ekkor:
```
for i in myrange(2, 10): print(i*i)
```
Mivel a generátor nem tárolja el az elemeket egyszerre, ezért akár végtelen generátort is csinálhatunk:
```
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. SageMath alapok

A Sage szintaktikája majdnem teljesen megegyezik a Pythonéval. A legfőbb különbség, hogy a hatványozás, amelyet Pythonban két csillaggal (`**`) írunk, pl. `2**3 == 8`, az Sage-ben kalap (`^`), pl. `2^3 == 8`. (Megjegyzés: Pythonban a `^`-jel a bitenkénti kizáró vagyot (XOR) jelenti, erre Sage-ben a kettős `^^` jelet használhatjuk.)
A Sage, ha elindítjuk, értelmezőként (interpreter) működik, azaz egyenként várja a parancsokat, és rögtön visszaadja az eredményt. Használhatjuk egyszerű számológépként, például:

```
sage: 1+1
2
sage: 2^10
1024
sage: factorial(33)
8683317618811886495518194401280000000
```

Itt a '`sage:`' felirat a *prompt*, azaz azután írhatjuk be a parancsot.
Az előző eredményre aláhúzásjellel (`_`) hivatkozhatunk, az azelőttire pedig két aláhúzásjellel (`__`) (ez háromig megy: `___`). Például:

```
sage: 2^3
8
sage: _ + 1
9
sage: (_ + 1) * __
80
```

Összetettebb programokat külön fájlba érdemes írni, és beolvasni Sage-be a `load()` függvénnyel. A Sage programok kiterjesztése `.sage`. (A fájl elkészítéséhez tetszőleges szövegszerkesztő program használható. Ha az nem támogatja a Sage-et, használhatjuk a Python szintaxiskiemelését is.)
Például ha a `new.sage` fájl tartalma a következő:

```
L = [1,2,5,10]
for n in L:
   print(n, 2^n, factorial(n))
```

akkor azt így tudjuk beolvasni:

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

(A `load()`-ban az elérési utat az aktuális könyvtárhoz képest kell megadni, azaz ahonnan a Sage-et elindítottuk. Az aktuális könyvtárat, ha nem tudjuk, a '`pwd`' paranccsal kérdezhetjük le.)
A Sage-ben – ellentétben a Python-nal – kérdőjellel (`?`) kérhetünk információkat egy függvényről, például (a súgóból kilépni a `q` billentyűvel lehet):

```
sage: sqrt?
sage: solve?
```

3.1. Szimbolikus számítás

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

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

Ez így látszólag nem túl hasznos: csak visszaadta, amit beírtunk. De nem azért, mert nem tud osztani. 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 (`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, méghozzá tetszőleges pontosságig:

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

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

```
sage: sqrt(2.)
1.41421356237310
```

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

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

Ez megint nem csinált semmit, de most azért nem, mert nem mondtuk meg, hogy mit szeretnénk vele csinálni. Például beszorozhatjuk, vagy ellenkezőleg, szorzattá alakíthatjuk a kifejezést:

```
sage: expand((x+1)^2)
x^2 + 2*x + 1
sage: expand((x+1/x)^8)
x^8 + 8*x^6 + 28*x^4 + 56*x^2 + 56/x^2 + 28/x^4 + 8/x^6 + 1/x^8 + 70
sage: factor(x^4 - 1)
(x^2 + 1)*(x + 1)*(x - 1)
```

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

```
sage: var("a,b")
(a, b)
sage: expand(a^2*(a-b)^3)
a^5 - 3*a^4*b + 3*a^3*b^2 - a^2*b^3
```

(Ha nem szeretnénk választ kapni, akkor írjunk pontosvesszőt az utasítás után, pl. '`var("a,b");`'.)

Fontos, hogy ne keverjük össze a szimbolikus változót a közönséges (Python-féle) változóval: a sima változónak mindig van egy konkrét értéke (pl. `5`), míg a szimbolikus változónak az értéke maga a szimbólum (pl. `x`).

```
sage: n = 5      # n egy változó
sage: n^2
25
sage: type(n)
<class 'sage.rings.integer.Integer'>
sage: var("n");  # n egy szimbolikus változó
sage: n^2
n^2
sage: type(n)
<class 'sage.symbolic.expression.Expression'>
```

(És vegyük észre azt is, hogy Sage-ben az egész számok típusa (`Integer`) nem ugyanaz, mint Python-ban (`int`). A Sage típusairól később lesz szó.)

Szintén ne keverjük össze a szimbolikus változókat a *szimbolikus konstans*okkal: `pi`, `e` stb., ezek ugyanis konkrét matematikai értékeket jelentenek. De ezekkel is lehet szimbolikusan számolni:

```
sage: x/4, pi/4     # eddig látszólag nincs különbség
(1/4*x, 1/4*pi)
sage: tan(x/4), tan(pi/4)    # de most már lesz
(tan(1/4*x), 1)
sage: N(e)   # szimbolikus konstansoknak van numerikus értéke
2.71828182845905
```

```
sage: N(x)   # 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
```

Ha egy beépített szimbolikus konstanst "elrontunk", akkor a `reset()` függvénnyel lehet visszaállítani:

```
sage: log(e)
1
sage: var("a,b,c,d,e");  # hoppá!
sage: log(e)
log(e)
sage: reset("e")
sage: log(e)
1
```

3.2. Matek a Sage-ben

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

`abs(x)`	$\lvert x\rvert$
`conjugate(x)`	$\overline{x}$
`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
```

Matematikai kifejezéseken, ahogy korábban már láttuk, a Sage különféle átalakításokat tud végezni (pl. `expand()`). Egyes műveleteket kétféle formában is megadhatunk: önálló függvényként, pl. `expand(...)`, vagy tagfüggvényként, pl. `(...).expand()`. (Ez egyébként igaz sok más függvényre is, pl. `4.sqrt() == 2`.)

Lássunk még néhány átalakítást (ne feledjük, '_' az előző eredményt jelenti):

```
sage: var("x,y,n");
sage: ((x+y)^2).expand()
x^2 + 2*x*y + y^2
sage: _.factor()
(x + y)^2
sage: ((x^2 - y^2) / (x - y)).simplify_full()
x + y
sage: (cos(x)^2 + sin(x)^2).simplify_full()
1
sage: (factorial(n+3) / factorial(n)).simplify_full().factor()
(n + 3)*(n + 2)*(n + 1)
sage: f = x^2 + y*x + 2
sage: f.subs(x = 2)
2*y + 6
sage: f.subs(x = sqrt(x))
sqrt(x)*y + x + 2
sage: f.subs(x = y, y = x)
x*y + y^2 + 2
sage: (x*y - 2*x^2 + x - y + x^2*y).collect(x)
x^2*(y - 2) + x*(y + 1) - y
sage: _.collect(y)
-2*x^2 + (x^2 + x - 1)*y + x
```

A Sage-ben összegezni a `sum()` függvénnyel lehet. Ezt kétféleképpen lehet használni. Az egyszerűbb (Python-ból örökölt) változatban egyszerűen meg kell adni egy listát (vagy egyéb sorozatot):

```
sage: sum([10,20,30])
60
sage: sum(range(10))
45
```

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

```
sage: var("n,k");
sage: sum(k^2, k, 1, 10)
385
sage: sum(k, k, 1, n).factor()
1/2*(n + 1)*n
sage: # végtelenig (oo == Infinity):
sage: sum(1/2^n, n, 0, oo)
2
sage: # szorzat ("produktum"):
sage: product(k^2, k, 1, n)
factorial(n)^2
```

$$\sum_{k=1}^{10} k^2 = 1^2 + 2^2 + \ldots + 10^2 = 385$$

$$\sum_{k=1}^{n} k = 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

$$\sum_{n=0}^{\infty} \frac{1}{2^n} = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots = 2$$

$$\prod_{k=1}^{n} k^2 = 1^2 \cdot 2^2 \cdot 3^2 \cdot \ldots \cdot n^2 = (n!)^2$$

Egyenleteket megoldani szimbolikusan a `solve()` függvénnyel lehet:

```
sage: var("x,y");
sage: solve(x^3 + 3*x^2 == 2*x + 2, x)
[x == -sqrt(2) - 2, x == sqrt(2) - 2, x == 1]
sage: solve([x + y == 6, x - y == 4], x, y)
[[x == 5, y == 1]]
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)]
```

A változókra feltételeket is támaszthatunk az `assume()` függvénnyel:

```
sage: solve(x^3 == 1, x)
[x == 1/2*I*sqrt(3) - 1/2, x == -1/2*I*sqrt(3) - 1/2, x == 1]
sage: assume(x, "real")
sage: solve(x^3 == 1, x)
[x == 1]
sage: forget()   # assume()-ok törlése, azaz x megint bármi lehet
sage: sqrt(x^2)
sqrt(x^2)
sage: assume(x, "real")
sage: sqrt(x^2)
abs(x)
sage: assume(x > 0)
sage: sqrt(x^2)
x
sage: forget()
sage: var("x,y");
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 az egészek között már csak véges sok megoldás van:
sage: assume(x, y, "integer")
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)]
```

Nem minden egyenletet lehet szimbolikusan megoldani. Például az ötödfokú egyenletnek nem létezéik általános megoldóképlete, ezért nem is oldja meg a Sage:

```
sage: solve(x^5 + x + 3, x)  # az "== 0"-t nem kötelező kiírni
[0 == x^5 + x + 3]
```

Megoldhatjuk viszont numerikusan, a `roots()` tagfüggvénnyel, megadva a számhalmazt, pl. $\mathbb{R}$ = `RR`, $\mathbb{C}$ = `CC` (a megoldás melletti másik szám a multiplicitás, azaz `1` = egyszeres gyök):

```
sage: (x^5 + x + 3).roots(x, ring=RR)
[(-1.13299756588507, 1)]
```

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

```
sage: var("n");
sage: limit(2*n/(3*n + 1), n = oo)
2/3
sage: limit(1/(x-1), x = 1, dir='-')
-Infinity
sage: diff(x^3 + sin(2*x), x)
3*x^2 + 2*cos(2*x)
sage: diff(x^3 + sin(2*x), x, 2)
6*x - 4*sin(2*x)
sage: integral(3*x^2 + 2*cos(2*x), x)
x^3 + sin(2*x)
sage: integral(sin(x), x, 0, pi)
2
```

$$\lim_{n\to\infty} \frac{2n}{3n+1} = \frac{2}{3}$$

$$\lim_{x\to 1^-} \frac{1}{x-1} = -\infty$$

$$\big(x^3 + \sin(2x)\big)' = 3x^2 + 2\cos(2x)$$

$$\big(x^3 + \sin(2x)\big)'' = 6x - 4\cos(2x)$$

$$\int \big(3x^2 + 2\cos(2x)\big)\, dx = x^3 + \sin(2x) + C$$

$$\int_0^{\pi} \sin x\, dx = 2$$

Vektorokat a `vector()` függvénnyel lehet létrehozni:

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

Mátrixokat pedig a `matrix()`-szal:

```
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.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 természetesen mindez szimbolikusan is működik, például így lehet egy általános 2×2-es mátrixszal 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 egyenletrendszert többféleképpen is megoldhatunk:

```
sage: A = matrix([[0,1,-3], [4,5,-2], [2,3,-1]]); 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]]
```

4. Szerző

Ezt a jegyzetet Uray M. János írta, részben felhasználva Nagy Ádám jegyzetét:
https://compalg.elte.gitlab-pages.hu/dimoa-web/tematika/dimoa
Hibákat, észrevételeket itt lehet jelezni: uray.janos@inf.elte.hu.