Hol találunk Python-t?

- Bejelentkezés `tarski.math.bme.hu`-ra; `leibniz`-en egy terminálból pl. így:

  ```
  ssh -Y tarski
  ```

  aztán `spyder3` vagy egy terminálban `ipython3`.
- Saját windows-os gépen: `http://wiki.math.bme.hu/view/AnacondaInstall` és aztán `spyder3`.
- `https://colab.research.google.com/` vagy `https://cocalc.com`. Regisztrálni kell, de szép jupyter notebook-ot kapunk.
- `https://sagecell.sagemath.org/`, válasszuk a Python-t. (Ez csak végszükségben javasolt megoldás.)

Olvasnivalók

- `http://math.bme.hu/~asimon/info2/python.pdf` (angol) jegyzet legújabb változata
- `http://wiki.math.bme.hu/view/Informatika2-2021` (tavalyi jegyzetek & egyebek)
- Wentworth &al., How to think like a computer scientist
- Hogyan gondolkozz úgy, mint egy informatikus: tanulás Python3 segítségével (Az előző egy korábbi változatának magyar fordítása.)

Gyakorlatok laborra:
`http://math.bme.hu/~asimon/info2/pytgyak.pdf`

Python nulláról

Python mint számológép

- A Python interpreter promptja így fest: `>>>` vagy `In [84]:`
- Ide lehet Python parancsokat írni, az interpreter végrehajtja őket, és visszaadja az eredményt (ha van).
- Jupyter notebook-ban `Ctrl+Enter`-t kell nyomni a parancs végrehajtásához.
- Példák:

```
Python 3.11.9 (main, Apr 17 2024, 00:00:00) [GCC 13.2.1 2024
Type "help", "copyright", "credits" or "license" for more in
>>> 2+3
5
>>> 2**3
8
>>> 2**3 == 8
True
>>> (123**13) % 13 == 123 % 13 #kis Fermat miatt
True
```

Python mint számológép

`#`-tól a sor végéig minden *comment*; vagyis Pythonban `#` olyan mint LaTeX-ben %.

Python mint számológép

Más típusú értékek is vannak, mint egész számok (`int`), pl.:

```
>>> 3.14 #lebegőpontos szám (float)
3.14

>>> 'This is a string' #string
'This is a string'

>>> [1, 2, 'Hello', 3.5] #lista (list)
[1, 2, 'Hello', 3.5]

>>> [1, 2, 'Hello', 3.5][2] #aminek ez a 2. eleme
'Hello'
```

Python mint számológép

- Ezeket az értékeket változókban lehet tárolni. Például:

```
>>> my_first_var = 2**3
```

- A változók nevei betűket, számokat és a _ karaktert tartalmazhatják (de nem kezdődhetnek számokkal).
- A fenti értékadás után `my_first_var` értéke `8` lesz, amíg meg nem változtatjuk:

```
>>> my_first_var
8
>>> 2 * my_first_var
16
```

Python mint programozási nyelv

- Miért jó, hogy értékeket lehet tárolni változókban?
- Nem elsősorban azért, mert így nem kell a program különböző pontjain újra és újra leírni, hogy (mondjuk) `[1, 2, 'Hello', 3.5]`.
- Hanem mert egy programban tipikusan sok különböző értékkel akarunk csinálni valamit, és a "csinálást" elég egyszer leírni, ha az a változón (és nem annak konkrét értékein) hajtódik végre.
- Persze ehhez az is kell, hogy a valtozó felvegye azokat az értékeket.

Ciklus

Ehelyett:

```
>>> print(1**2)
1

>>> print(2**2)
4

>>> print(3**2)
9

>>> print(4**2)
16
```

while ciklus

sokkal praktikusabb a következő:

```
>>> number = 1
>>> while (number < 5):
...     print(number ** 2)
...     number = number + 1
...
1
4
9
16
```

Azt kellett megadnunk, hogy

- milyen értékekkel
- mit akarunk csinálni

while ciklus

Ezért mindkettőt könnyű megváltoztatni. Itt pl. az előbbit változtatjuk:

```
>>> number = 1
>>> while (number < 15):
...     print(number ** 2)
...     number = number + 3
...
1
16
49
100
169
```

`while` ciklus

A ciklust nem csak kényelmetlen, de sokszor lehetetlen is volna különálló `print()`-ekkel kiváltani: pl. mit tennénk, ha az a `15` a program futása során derül csak ki.

`while` ciklus

A `while` ciklus szintaxisa:

`while` *feltétel*:
    `tedd_ezt`
    `...`
    `tedd_azt`

Ez újra és újra végrehajtja a ciklus *testét* ameddig a feltétel igaz. A test addig tart, ameddig a `while...`-énál nagyobb indentálás.

`while` ciklus

```
number = 1
while (number < 15):
    print(number ** 2)
    number = number + 3
print('Done')
```

```
1
16
49
100
169
Done
```

Ezért van az, hogy a `print('Done')` csak egyszer, a legvégén hajtódik végre.

`print()`

Fontos különbség van

```
>>> 42
42
```

és

```
>>> print(42)
42
```

között: az első visszaad egy értéket (amit az interpreter szokása szerint kinyomtat), a második semmit sem ad vissza, de a `print()` *mellékhatása*, hogy kinyomtatja az argumentumát. Tehát a látvány ugyanaz, de *csak a látvány*.

print()

Itt látható a különbség:

```
>>> a = 42
>>> print(a)
42
```

szemben ezzel:

```
>>> a = print(42)
42
>>> print(a)
None
```

A (képernyőre való) `print`-elés csak a felhasználó tájékoztatására szolgál.

for ciklus

Másfajta, gyakran kényelmesebb, ciklus: a `for` ciklus. A fenti első `while` ciklus `for` ciklussal:

```
>>> number = 1
>>> while (number < 5):
...     print(number ** 2)
...     number = number + 1
...
1
4
9
16
```

```
>>> for number in range(1,5):
...     print(number ** 2)
...
1
4
9
16
```

Sokkal elegánsabb!

for ciklus

A második:

```
>>> number = 1
>>> while (number < 15):
...     print(number ** 2)
...     number = number + 3
...
1
16
49
100
169
```

```
>>> for number in range(1,15,3):
...     print(number ** 2)
...
1
16
49
100
169
```

`for` nem csak számsorozatokon tud iterálni, hanem mindenféle "iterálható" értéken[1]: `list`án, `str`ingen, egy szövegfile sorain, adatbázis rekordjain, ...

[1] *iterable*

for ciklus

A fenti range-ek nem listák, de azokká konvertálhatók a list() függvény segítségével: pl.

```
>>> list(range(1,15,3))
[1, 4, 7, 10, 13]
```

Utolsó példa a kétféle ciklus összehasonlítására: egy számokból álló lista tagjainak szorzata. Ha l = [4,2,5,9,10], akkor

```
>>> prod = 1
>>> i = 0
>>> while i < len(l):
...     prod = prod * l[i]
...     i = i + 1
...
>>> prod
3600
```

```
>>> prod = 1
>>> for n in l:
...     prod = prod * n
...
>>> prod
3600
```

`break`, `continue`

- Mindkét fajta ciklusból kiléphetünk idő előtt (`break`), vagy azonnal a következő iterációra (`while`-ban a feltétel kiértékelésére, `for`-nál a következő értékre) ugorhatunk (`continue`).
- De ezeknek nem sok értelme lenne, ha feltétel nélkül hajtódnának végre.
- Tipikus példák (mindjárt jönnek): valamilyen gyűjtemény, pl. egy lista tagjaival kell valamit csinálnunk, de
  - csak addig, amíg nem akad köztük valami speciális tulajdonságú (`break`)
  - csak a speciális tulajdonságúakkal kell megcsinálnunk ezt a valamit (`continue`)
- Tehát kell egy konstrukció, ami egy feltételtől függően hajt (vagy nem hajt) végre valamit. Esetleg különböző dolgokat hajt végre.

`if` parancs

```
if feltétel:
    tedd_ezt_ha
    ...
    tedd_azt_ha
```

a testében (`tedd_ezt_ha`,...,`tedd_azt_ha`) szereplő összes utasítást végrehajtja, de csak ha a *feltétel* igaz;

```
if feltétel:
    tedd_ezt_ha
    ...
    tedd_azt_ha
else:
    tedd_ezt_ha_nem
    ...
    tedd_azt_ha_nem
```

ugyanazt teszi, ha igaz a *feltétel*, és az `else` ág testében szereplő utasításokat hajtja végre ha nem.

if parancs

Például:

```
>>> if 2<3:
...     print('OK')
...
OK
>>> if 3<2:
...     print('OK')
...
>>> if 3<2:
...     print('OK')
... else:
...     print('Not OK')
...
Not OK
```

Példa `continue` használatára. Tegyük fel, hogy egy számlista páratlan elemeinek szorzatát akarjuk kiszámolni. (`l` továbbra is [4, 2, 5, 9, 10].)

```
>>> prod = 1
>>> for n in l:
...     if n%2 == 0: #ha n páros
...         continue #vegyük a lista következő elemét
...     prod = prod * n
...
>>> prod
45
```

Feladat
Csináljuk meg ezt `continue` használata nélkül.

break, continue

Példa `break` használatára. Tegyük fel, hogy egy számlista azon legnagyobb kezdőszelete elemei szorzatát akarjuk kiszámolni, amiben még nem szerepel páratlan szám. (`l` továbbra is [4, 2, 5, 9, 10].)

```
>>> prod = 1
>>> for n in l:
...     if n%2 == 1: #ha n páratlan
...         break    #azonnal lépjünk ki!
...     prod = prod * n
...
>>> prod
8
```

Függvények

Korábban volt ez a példánk:

```
>>> prod = 1
>>> for n in l:
...     prod = prod * n
...
>>> prod
3600
```

ami az `l` listabeli számokat szorozta össze (és tette az eredményt a `prod` változóba). Ha nem csak egyszer akarjuk egy lista elemeit összeszorozni, akkor érdemes ezt egy újrahasznosítható, névvel ellátott programmá ("függvénnyé") alakítani.

Függvények

```
>>> def product(l):
...     result = 1
...     for n in l:
...         result = result * n
...     return result
...
```

és ezt aztán használhatjuk (*hív*hatjuk) így:

```
>>> product([4,2,5,9])
360

>>> product([4,2,5,9,10])
3600
```

Függvények

Mit csináltunk és miért jó ez?

- Az eredeti programunkból *kiabsztraháltuk* `l`-et. Pontosan úgy, ahogy $\sin\frac{\pi}{3}$-ból $\frac{\pi}{3}$ kiabsztrahálásával megkapjuk a sin függvényt.
- És ahogy ez utóbbit sokszor $\sin x$ függvényként szokás emlegetni, de tudjuk, hogy a $\sin y$ is ugyanaz a függvény, egy Python függvény definíciójában sem számít a függvény *paraméter*ének neve (ami itt most `l`).
- `l` és `result` *lokális változók*, ami azt jelenti, hogy még ha vannak is ilyen nevű változóink a függvénydefiníción kívül, azok értékei visszaállnak a függvény visszatérésekor. (Ez fontosabb, mint amilyennek esetleg látszik.)
- Egy függvény újrahasznosítható: ha a programunk több helyen is számolja listák szorzatát, bemásolhatnánk a kódot minden ilyen helyre, de akkor pl. megváltoztatni is sok helyen kellene ha mondjuk hibát találunk benne.

Függvények

Itt látszik, hogy mit tudnak a lokális változók:

```
>>> a = 1
>>> b = 2
>>> def fun(a):
...     b = a
...     return b
...
>>> fun(42)
42
>>> a
1
>>> b
2
```

A függvényeken kívül definiált változók *globális* változók. (Ezek lehetőleg kerülendők.)

Függvények

A függvénydefiníció szintaxisa:

`def` *name(parameter1,parameter2,...)*:

```
    tedd_ezt
    ...
    tedd_azt
```

A függvény testében szerepelhet egy vagy több

`return` *érték*

aminek hatására a függvény visszatér az

érték

eredménnyel. Ha nem `return` hatására tér vissza a függvény, akkor `None` az eredménye (vagyis nincs neki). Ezt teszi pl. a `print()` függvény. Az ilyen függvényeket a *mellékhatás*aik miatt hívjuk meg.

Függvények

Példa egy ilyen függvényre:

```
def show(arg):
    print(arg, 'típusa', type(arg))
    if isinstance(arg,list) or isinstance(arg,str):
        print(len(arg), 'hosszú.')
    else:
        if isinstance(arg,int):
            print('Páratlan.' if arg%2 == 1 else 'Páros.')
        else:
            print('Többet nem tudok mondani róla')
```

Függvények

```
>>> show(['Hell', 'o', 12])
['Hell', 'o', 12] típusa <class 'list'>
3 hosszú.
>>> show('Hello')
Hello típusa <class 'str'>
5 hosszú.
>>> show(3)
3 típusa <class 'int'>
Páratlan.
>>> show(3.14)
3.14 típusa <class 'float'>
Többet nem tudok mondani róla
```

Függvények

Újdonságok:

- `print()`-nek több argumentuma is lehet, ezeket szóközzel elválasztva nyomtatja ki.

```
>>> print(3, 'cica')
3 cica
```

- Pythonban minden objektumnak van típusa, amit egyrészt meg lehet kérdezni a `type()` függvény segítségével, de azt is meg lehet tudni, hogy egy objektum bizonyos típusú-e.

```
>>> type(1)
<class 'int'>
>>> isinstance(1,int)
True
>>> isinstance(1,list)
False
```

Függvények

- Feltételekből (azaz `bool` típusú értékekből) Boole-műveletek (itt most épp `or`) segítségével újabb `bool`-okat lehet gyártani.
- `list`áknak és `str`ingeknek van hosszuk, amit a `len()` függvény mér.
- Használtuk az `if` *kifejezés*t, ami különbözik a korábbi `if` *parancs*tól; ez az első (az `if` előtti) argumentumát adja vissza ha a második (az `if` és `else` közti) argumentuma `True`, máskülönben a harmadikat (az `else` utánit).

```
>>> 'Igen' if 2 <= 3 else 'Nem'
'Igen'

>>> 'Igen' if 2 == 3 else 'Nem'
'Nem'
```

Függvények

Vagyis az `if` *kifejezés* valamilyen feltételtől függő *érték*et ad vissza (ezért hívják kifejezésnek). Ez legalább két esetben hasznos.

- `maximum = b if a < b else a`

  sokkal tömörebb, mint a vele ekvivalens

```
if a < b:
    maximum = b
else:
    maximum = a
```

Feladat
Írjuk át a `show()` függvényt úgy, hogy ne használjon `if` kifejezést.

- Anonim függvényekben (`lambda`-kban).

Anonim függvények (`lambda`-k)

Ezek "egyszer használatos" függvények. Egy nem tipikus példa a használatukra:

```
>>> (lambda x : x ** 2)(3)
9
```

Tipikus példa: amikor egy függvény argumentumként vár egy másik függvényt, pl. egy rendező függvénynek meg kell mondanunk, hogy a rendezendő értékeknek hogyan számítsa ki azt a jellemzőjét, ami alapján rendeznie kell (ld. itt!).

Egy `lambda` teste egyetlen kifejezésből áll, és ennek az értékét adja vissza. Tehát döntésre csak az `if` *kifejezés*t használhatjuk, a parancsot nem.

```
>>> (lambda x : x**3 if x > 0 else -x**3)(-2)    #/x^3/
8
```

vagy akár

```
>>> (lambda x : (x if x > 0 else -x)**3)(-2)     #/x/^3
8
```

listák (és más sorozatok) indexelése

Ezt már láttuk:

```
>>> numbers = list(range(17))
>>> numbers
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
>>> numbers[10]
10
```

De [] többet tud ennél.

- index lehet negatív: ilyenkor hátulról számolódik (1-el kezdve) az index.

  ```
  >>> numbers[-3]
  14
  >>> numbers[-3]  == numbers[len(numbers)-3]
  True
  ```

listák (és más sorozatok) indexelése

- Nem csak elemeket, de szeleteket (*slices*) is megadhatunk:

```
>>> numbers[10:15]  #a 10.-től a 14.-ig
[10, 11, 12, 13, 14]
```

- Ilyenkor első és utolsó indexet nem kell megadnunk:

```
>>> numbers[:10]  #első 10
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> numbers[15:]  #a 15.-től kezdve mindegyik
[15, 16]
```

- Lépésközt (*stride*) is megadhatunk:

```
>>> numbers[1:8:2]  #az első 4 páratlan
[1, 3, 5, 7]
>>> numbers[::3]  #minden 3.
[0, 3, 6, 9, 12, 15]
```

listák (és más sorozatok) indexelése

- Szeletek negatív indexekkel:

```
>>> numbers[:-10] #hátulról a 10.-ig
[0, 1, 2, 3, 4, 5, 6]
>>> numbers[-10:] #hátulról a 10.-től (= az utolsó 10)
[7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
>>> numbers[-10:-3]  #az utolsó 10, kivéve az utolsó 3-at
[7, 8, 9, 10, 11, 12, 13]
>>> #ugyanaz: elfelejtjük az utolsó 3-at,
>>> #aztán  vesszük az utolsó 7-et
>>> numbers[:-3][-7:]
[7, 8, 9, 10, 11, 12, 13]
>>> numbers[-10:][:7] #ugyanaz: utolsó 10-ből, az első 7
[7, 8, 9, 10, 11, 12, 13]
```

listák (és más sorozatok) indexelése

- Lépésköz is lehet negatív...

```
>>> numbers[::-1]  #mindegyik, de visszafelé
[16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> numbers[::-2]  #hátulról minden második
[16, 14, 12, 10, 8, 6, 4, 2, 0]
```

- ...de akkor *start* és *end* (amiket továbbra is elölről számolunk) felcserélődnek:

```
>>> # 4.-től az elsőig (de az már nem,
>>> # mert 'jobbról' nyílt i.v., mint mindig)
>>> numbers[4:1:-1]
[4, 3, 2]
>>> numbers[4::-1]  # első 5 visszafelé
[4, 3, 2, 1, 0]
```

listák (és más sorozatok) indexelése

- Mindezek működnek stringekre is:

```
>>> s = 'abcdefgh'
>>> s[3]
'd'
>>> s[-3]
'f'
>>> s[5:]
'fgh'
>>> s[5::-1]
'fedcba'
```

- De fontos különbség, hogy stringeket nem lehet megváltoztatni:

```
>>> s[3]='x'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

listák (és más sorozatok) indexelése

- listákat viszont igen:

```
>>> numbers = list(range(15))
>>> numbers
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> numbers[0]=-10
>>> numbers
[-10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> numbers[1:4] = [11,22,33]
>>> numbers
[-10, 11, 22, 33, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> numbers[-1:-4:-1] = [111,222,333]
>>> numbers
[-10, 11, 22, 33, 4, 5, 6, 7, 8, 9, 10, 11, 333, 222, 111]
```

listák (és más sorozatok) indexelése

- Sőt:

```
>>> numbers = list(range(15)) ; numbers[1::4]
[1, 5, 9, 13]
>>> numbers[1::4] = [-1,-5,-9,-13] ; numbers
[0, -1, 2, 3, 4, -5, 6, 7, 8, -9, 10, 11, 12, -13, 14]
```

  Mert ez az a lista, amire

```
>>> numbers[1::4] == [-1,-5,-9,-13]
True
```

- A szelet új értéke az eredetitől eltérő hosszúságú is lehet:

```
>>> numbers = list(range(15))
>>> numbers[1:10] = [] ; numbers
[0, 10, 11, 12, 13, 14]
>>> numbers[1:3] = list(range(100,106)) #az [1,3) iv helyébe
>>> numbers
[0, 100, 101, 102, 103, 104, 105, 12, 13, 14]
```

listák (és más sorozatok) indexelése

(Itt több parancs van egy sorban, pontosvesszővel elválasztva. Ezek épp úgy egymás után hajtódnak végre, mint ha különböző sorokban lennének, de csak az utolsó értékét kapjuk vissza.)

- Feltéve, hogy nincs megadva lépésköz:

```
>>> numbers = list(range(15))
>>> numbers[1::4]
[1, 5, 9, 13]
>>> numbers[1::4] = [-1,-5,-9]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: attempt to assign sequence of size 3 to extended
```

mert ennek nem lenne értelme.

listák (és más sorozatok) indexelése

Feladat
Mi a különbség `numbers[1] = [True]` és `numbers[1:2] = [True]` között? Ekvivalens-e valamelyikük `numbers[1] = True`-val?

`list`agenerálás[2]

- Ha `l` "iterálható" (mint a `for` ciklusban) (pl. egy `list`a, egy `str`ing vagy egy `range`), akkor

  `[f(x) for x in l]`

  azt a `list`át adja vissza, aminek *i*. eleme `f` alkalmazva `l` *i*. elemére.

  ```
  >>> [x/2 for x in range(10)]
  [0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5]
  ```

- Nem kell minden elemet megtartanunk:

  ```
  >>> [x/2 for x in range(10) if x%2 == 0]
  [0.0, 1.0, 2.0, 3.0, 4.0]
  ```

[2] `list` comprehension

listagenerálás

- Ezek (természetesen) egymásba ágyazhatók:

```
>>> [[str(i)+j for j in 'abc'] for i in range(3)]
[['0a', '0b', '0c'], ['1a', '1b', '1c'], ['2a', '2b', '2c']]
```

  Itt használtuk a `str()` függvényt, ami az argumentuma `str`inggé konvertált változatát adja vissza, és azt, hogy `str`ingek összeafűzhetők `+`-al.

Változó-kicsomagolás (első közelítés)

- A következő kifejezés értéke

```
>>> 2+3, 2*3, 2**3
(5, 6, 8)
```

nem három különböző objektum, hanem egy `tuple` (rendezett n-es, mint a lista, csak ez nem módosítható). Ez itt látszik:

```
>>> a = 2+3, 2*3, 2**3
>>> a
(5, 6, 8)
>>> type(a)
<class 'tuple'>
```

- De a "koordinátáira" cincálható, pl. így:

```
>>> a, b = 2+3, 2*3
>>> a
5
>>> b
6
```

Változó-kicsomagolás (első közelítés)

- Ennek egyik hasznos következménye, hogy

```
>>> a = 5
>>> b = 10
```

helyett ezt is írhatjuk:

```
>>> a, b = 5, 10
```

- A kettő nem teljesen ugyanaz, mert a második esetben az értékadás párhuzamosan történik.
- Következésképp ezt használhatjuk két változó értékének felcserélésére:

```
>>> a, b = b, a
>>> a, b
(10, 5)
```

amihez amúgy egy segédváltozóra lenne szükség:

```
>>> temp = a ; a = b; b = temp
>>> a, b
(5, 10)
```

metódusok

- Metódusok csak jóval később lesznek hivatalosan bevezetve, de használni már most is szeretnénk őket.
- Gondolhatunk rájuk mint furcsán használt függvényekre.
- Például ahelyett, hogy `upper('abc')` hívással állítanánk elő `'abc'` nagybetűs változatát, ezt írjuk:

  ```
  >>> 'abc'.upper()
  'ABC'
  ```

  mert az `.upper()` `str`ingek egy metódusa.
- Ez azt jelenti, hogy a pont előtt `str`ingnek kell állnia. Erre (a példában `'abc'`-re) hívtuk meg a metódust.

metódusok

- Más példa: az `.append()` `list`ák egy metódusa, ami az argumentumaként megadott értéket ahhoz a listához adja, amire meghívtuk:

```
>>> l = [1,2,3]
>>> l.append(100)
>>> l
[1, 2, 3, 100]
```

  Szemben `.upper()`-rel, ezt nem az értékéért hívjuk (ami nincs neki), hanem a mellékhatásáért.
- Képzelhetjük úgy, hogy mondjuk az

```
obj1.m(obj2, obj3)
```

  az

```
m(obj1, obj2, obj3)
```

  függvényhívás közeli rokona, csak épp `obj1` típusa (osztálya) és `m` speciális kapcsolatban állnak.

Modulok

- Az `import` kulcsszó használatával mindig valamilyen extra funkciókhoz való hozzáférhetőséget biztosítjuk a programunk számára.
- Például:

```
>>> import math
>>> math.sqrt(2), math.pi
(1.4142135623730951, 3.141592653589793)
```

- A `math` modulról így kaphatunk részletes tájékoztatást:

```
help(math)
```

- ami így kezdődik:

```
Help on module math:

NAME
    math
...
```

1. példa

Definiáljunk egy `repeats()` függvényt, melynek egyetlen argumentuma számok egy listája lesz, és amely `True`-t ad vissza, ha a lista valamely két egymást követő tagja egyenlő, és `False`-t egyébként. Pl.

```
>>> repeats([])
False
>>> repeats(range(10))
False
>>> repeats([1,2,1,4])
False
>>> repeats([1,2,1,4,4])
True
>>> repeats([1,2,1,1,4])
True
```

1. példa

```
def repeats(l):
    for i in range(1,len(l)):
        if l[i-1] == l[i]:
            return True
    return False
```

vagy

```
def repeats(l):
    if l == []: return False  #muszáj ellenőrizni a
                              #következő sor miatt
    last = l[0]
    for i in l[1:]:
        if i == last:
            return True
        last = i
    return False
```

2. példa

Definiáljuk a kétargumentumú `squares()` függvényt úgy, hogy `squares(m,n)` az m és n természetes számok közti négyzetszámokat adja vissza. Pl.

```
>>> squares(10, 15)
[]
>>> squares(9, 25)
[9, 16, 25]
```

```
def squares(m,n):
    return [i**2 for i in range(n+1) if m <= i**2 <= n]
```

ami nem túl hatékony,

2. példa

vagy

```
def squares(m,n):
    res = []
    i = isquare = 0  #ilyet is lehet
    while isquare <= n:
        if isquare >= m:
            res.append(isquare)
        i = i + 1
        isquare = i**2
    return res
```

ami szintén nem,

2. példa

vagy[3]

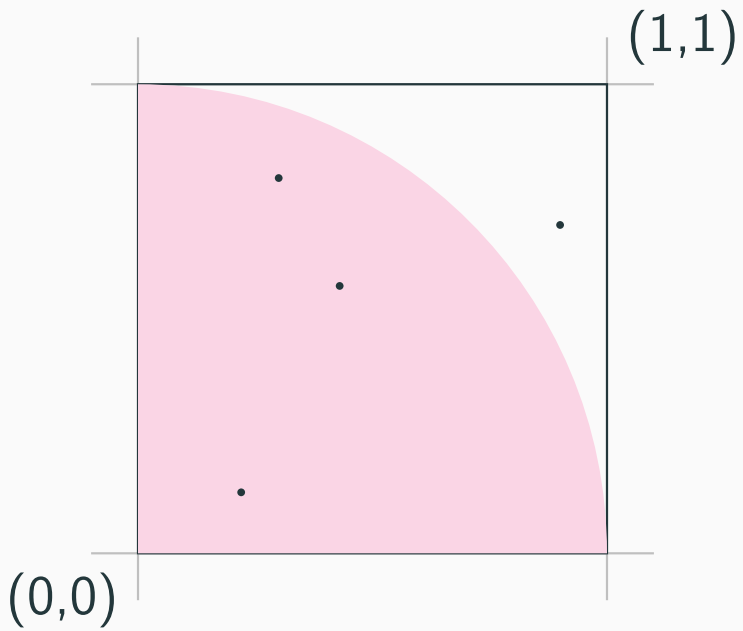
```
import math
def squares(m,n):
    return [i**2 for i in range(math.ceil(math.sqrt(m)),
    1+math.isqrt(n))]

>>> math.isqrt?
Return the integer part of the square root of the input.
>>> math.ceil?
Return the smallest integer >= its argument.
```

[3] i $\mathbb{N}$-re $m \leq i^2 \leq n \iff \sqrt{m} \leq i \leq \sqrt{n} \iff \lceil \sqrt{m} \rceil \leq i < 1 + \lfloor \sqrt{n} \rfloor$

3. példa

Definiáljunk egy egyargumentumú függvényt, ami π értékét közelíti úgy, hogy ha az argumentum `n`, akkor n tűt dob az egységnégyzet véletlenszerű pontjaiba, és az egységkör abba eső negyedébe érkezők és n hányadosának négyszeresét adja vissza.



3. példa

Az alább használt `random` modulbeli `random()` függvény egy $[0, 1]$-beli `float`-ot ad vissza. Mivel sokszor[4] használjuk, úgy importáljuk, hogy ne kelljen eléírni a modul nevét.

```
from random import random
def mcpi(n):  #Monte Carlo pi
    incircle = 0
    for _ in range(n):  #nem használjuk a ciklus-változót
        px,py = random(),random()
        if px ** 2 + py ** 2 <= 1:
            incircle += 1  #incircle inkrementálása
                           #ld. alább!
    return(4*incircle/n)
```

```
>>> mcpi(10) , mcpi(10 ** 3) , mcpi(10 ** 6), mcpi(10 ** 6)
(3.2, 3.116, 3.139392, 3.140912)
```

[4] kétszer

3. példa

A függvény definíciójában használt `incircle += 1` ekvivalens `incircle = incircle + 1`-gyel. Ez a rövidítés (in-place assignment) minden binér operációra működik: ha `o` binér operáció, `x` változó, és `v` értéke olyan, hogy `x o v`-nek van értelme (`o` alkalmazható `x` értékére és `v`-re), akkor `x o= v` ekvivalens `x = x o v`-vel. Pl.

```
>>> b = 3; b **= 2; b
9
```

mert `b **= 2` és `b = b ** 2` ekvivalensek. Hasonlóan:

```
>>> b = 14; b %= 3; b
2
```

További részletek

Egyszerű típusok

Típus	Leírás
`int`	egész szám
`float`	lebegőpontos szám
`complex`	komplex szám
`bool`	boolean (`True` és `False`)
`NoneType`	`None` (null érték)

Table 1: Néhány egyszerű típus

Egyszerű típusok

Komplex számok így írhatók: `a+bj`, ahol `a` és `b` `int` vagy `float`.

```
>>> (1+1j)**2
2j
```

`j` magában csak egy változónév:

```
>>> j**2 == -1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'j' is not defined
>>> 1j**2 == -1
True
```

Egyszerű típusok

Két `int` hányadosa akkor is `float`, ha a második osztja az elsőt!

```
>>> 2/2, type(2/2)
(1.0, <class 'float'>)
```

Ami azért is gond, mert a `float`-ok korlátozott pontosságúak, aminek néha meglepő következményei vannak:

```
>>> 18530201888518410 / 2
9265100944259204.0
```

Ezért, ha tudjuk, hogy egész szám lesz az eredmény, használjunk `//`-t `/` helyett:

```
>>> 18530201888518410 // 2
9265100944259205
```

Egyszerű típusok

Ha nem tudjuk, akkor is lehet ilyet:

```
x//y if x%y == 0 else x/y
```

Például:

```
>>> [x//3 if x%3 == 0 else x/3 for x in range(5,9)]
[1.6666666666666667, 2, 2.3333333333333335, 2.6666666666666665]
```

Hasznos lehet a `divmod()` függvény is, ami az `(x//y, x%y)` párt adja vissza.

Végül: egyszerű típus még a `NoneType` is; ez egyetlen értéknek a típusa, a `None`-nak, amit a semmit vissza nem adó függvények adnak vissza. (Később látni fogjuk egy másik használatát.)

Összetett típusok

- `str`ingek: egyelőre elég annyi, amennyit eddig tanultunk róluk, és hogy mindenféle idézőjelek közé (mint `'` és `"`) kell írni őket
- `list`ák: egyelőre ezekről is elég annyi, amennyit már tanultunk (beleértve az `.append()` metódust, amivel értékeket lehet egy `list`a végéhez adni)
- `tuple`-k: ld. a változó-kicsomagolásról szóló részt itt!
- és még lesz ilyen

Általánosított `bool` értékek

Az `if` (parancs vagy kifejezés) vagy `while`-beli feltétel rendszerint `bool` típusú, de más is lehet, mint a következő példákban:

Általánosított `bool` értékek

```
>>> if 0: print('Yes')
...
>>> if 10: print('Yes')
...
Yes
>>> if []: print('Yes')
...
>>> if [0]: print('Yes')
...
Yes
>>> if '': print('Yes')
...
>>> if 'nonempty string': print('Yes')
...
Yes
>>> if None: print('Yes')
...
```

`if` parancs

Mindent tudunk már róla, csak azt nem, hogy lehetnek `elif` ágai:

```
if feltétel_1:
    tedd_ezt_ha_1
    ...
elif feltétel_2:
    tedd_ezt_ha_2
    ...
else:
    tedd_ezt_ha_egyik_sem
    ...
```

ami ugyanazt teszi, csak tömörebb és világosabb, mint a következő:

if parancs

```
if feltétel_1:
    tedd_ezt_ha_1
    ...
else:
    if feltétel_2:
        tedd_ezt_ha_2
        ...
    else:
        tedd_ezt_ha_egyik_sem
        ...
```

if parancs

Például:

```
>>> for i in range(-5,26,10): #azaz [-5,5,15,25]:
...     if i<0:
...         print('negative')
...     elif i<=10:
...         print('small')
...     elif i<=20:
...         print('medium')
...     else:
...         print('big')
...
negative
small
medium
big
```

Feladat

Írjuk meg ugyanezt `elif` nélkül!

Binér operátorok és relációk

Operátorok és relációk	Leírás
`or`	Boole vagy
`and`	Boole és
`not`	Boole tagadás
`in`, `not in`	eleme reláció tesztelése
`is`, `is not`	azonosság tesztelése
`<`, `<=`, `>`, `>=`, `==`, `!=`	összehasonlítás
`+`, `-`	összeadás, kivonás
`*`, `/`, `//`, `%`	szorzás, osztás, hányados (`5//2==2`), maradék
`**`	hatványozás (`^` *nem* az!)

Table 2: Néhány binér operátor és reláció

Binér operátorok és relációk

Ha `R`, `S` binér relációk, akkor `x R y and y S z` helyett írhatjuk ezt:

`x R y S z`

Például

```
>>> 3 <= 4 > 2 < 10
True
```

Így megspórolhatunk néhány `and`-et.

Ciklusokról bővebben: `else`

Minkét fajta ciklust ki lehet egészíteni egy `else` ággal, ami akkor hasznos, ha `break`-et használunk a ciklus testében. Ez a kibővített `for` ciklus szintaxisa, de a `while` hasonlóan bővíthető:

```
for változó in iterálható:
    tedd_ezt
    ...
    tedd_azt
else:
    tedd_ezt_is
    ...
    tedd_azt_is
```

Az `else` ág csak akkor hajtódik végre, ha a ciklus normálisan (nem `break` végrehajtásával) ér véget.

Ciklusokról bővebben: else

Egy példa, ahol ez hasznos:

Tegyük fel, hogy számok egy listajából az első 7-el oszthatót akarjuk kinyomtatni, vagy ha nincs ilyen, akkor ezt a tényt.

```
numbers = [2, 5, 10, 14, 21, 35, 42, 51]


for n in numbers:
    if n % 7 == 0:
        print(n, "osztható 7-el")
        break
else:
    print("Nincs a listában 7-el osztható szám.")
```

```
14 osztható 7-el
```

Nem írhatjuk az utolsó `print()` utasítást símán a ciklus után, mert így akkor is végrehajtódna, ha volna a listában 7-el osztható szám.

Ciklusokról bővebben: `else`

`else` híján egy extra változóban kellene követni, hogy mi történik a ciklus testében:

```
numbers = [2, 5, 10, 14, 21, 35, 42, 51]
broke = False

for n in numbers:
    if n % 7 == 0:
        print(n, "osztható 7-el")
        broke = True
        break
if not(broke):
    print("Nincs a listában 7-el osztható szám.")
```

```
14 osztható 7-el
```

Ez nem csak csúnya, de extra hibalehetőség is, mert mi van, ha ez a kódrészlet egy nagyobb része, ami maga is használ `broke` nevű változót.

Ciklusokról bővebben: `else`

Amúgy ilyen esetekben sokszor jobban járunk, ha egy külön függvényt írunk és abban `return`-t használunk `break` helyett, és így kerüljük el azt, hogy a ciklus utáni utasítások végrehajtódjanak, ha nem kellene nekik.

Feladat
Írjunk `break` és `else` használata nélkül egy `first_divisible(numbers,d)` függvényt, ami kiírja a `numbers` számlista első olyan tagját, amely osztható d-vel, vagy ha nincs ilyen, akkor ezt a tényt. Például:

```
>>> nums = [2, 5, 10, 14, 21, 35, 42, 51]
>>> first_divisible(nums, 7)
14 osztható 7-al/el

>>> first_divisible(nums, 13)
Nincs a listában 13-al/el osztható szám.
```

Ciklusokról bővebben: `enumerate()`

Gyakran akarunk iterálni valamilyen gyűjtemény fölött úgy, hogy közben az épp aktuális tag sorszámára is szükségünk van. Erre a célra szolgál az `enumerate()` függvény, amit így lehet használni:

```
>>> for index, number in enumerate(range(10,15)):
...     print(index, number)
...
0 10
1 11
2 12
3 13
4 14
```

Ciklusokról bővebben: `enumerate()`

Ez sokkal tömörebb, mint ha mondjuk ezt írnánk:

```
index = 0
for number in range(10,15):
    print(index, number)
    index += 1
```

Ciklusokról bővebben: `enumerate()`

Egy extra (“opcionális”) argumentum használatával `enumerate()` 0-tól különböző egésztől kezd el számolni:

```
>>> for index, number in enumerate(range(97,107),1):
...     print(index, chr(number))
...
1 a
2 b
3 c
4 d
5 e
6 f
7 g
8 h
9 i
10 j
```

Ciklusokról bővebben: `enumerate()`

Az csak látszat, hogy itt két ciklus-változó (`index` és `number`) van. Ami itt történik, az csak a már ismert változó-kicsomagolás (ld. itt!) egy esete. `enumerate()` párok gyűjteményét adja vissza:

```
>>> list(enumerate(range(10,15)))
[(0, 10), (1, 11), (2, 12), (3, 13), (4, 14)]
```

azaz minden "fordulóban" egy párt kapunk, amit alkotóelemeire cincálunk: első tagját az `index`, a másodikat a `number` változóhoz rendeljük.

I/O

I/O: `input()`

Legnagyobb hiányosságunk: csak azokkal az adatokkal tudunk dolgozni, amelyek a program megírása során is rendelkezésre állnak. Ez matek (pl. Sage) programoknál nem olyan nagy gond, de általában igen.

Második legnagyobb hiányosságunk: csak minimális mennyiségű adatot tudunk használható formában közölni (a `print()` függvény segítségével).

Adatbevitel a billentyűzetről: az `input()` függvény `str`ingként adja vissza, amit a felhasználó begépel mielőtt megnyomná a Return billentyűt. Például:

```
x = int(input())
print(2*x)
```

Ez addig vár, míg be nem gépelünk egy számot és nem nyomunk utána Return-t; akkor az `int()` egésszé (mondjuk `42`-vé) alakítja a beolvasott `str`inget (`'42'`-t) és kinyomtatja a dupláját.

I/O: `input()`

`input()`-ot felhasználóbarátabbá tehetjük ha ún. *opcionális argumentum*ként megadunk egy prompt-ot (egy `str`inget, ami a megjelenik a képernyőn és tudatja a felhasználóval, hogy mit vár tőle a program).

```
x = int(input('Adjon meg egy egész számot: '))
print (2*x)
```

Feladat

Írjon egy programot, ami számokat kér egymás után, és ha a felhasználó `RETURN`-t nyom kétszer egymás után, akkor kinyomtatja az addig bevitt számok átlagát.

```
Írjon be egy számot: 4
Írjon be egy számot: 12
Írjon be egy számot: 5
Írjon be egy számot:
Az átlag 7.0
```

I/O: input()

Egy számlistát kényelmesebb lenne szóközökkel elválasztott listaként, egy lélegzettel bevinni. Ehhez az input() által visszaadott stringet kell manipulálni. Például:

```
is = input('Írjon be számokat szóközökkel elválasztva: ')
l = [int(i) for i in is.split()]
```

A .split() metódus a stringbeli "szavak" listáját adja vissza. Például:

```
>>> '12  23 42     135  '.split()
['12', '23', '42', '135']
```

Feladat

Írjon egy programot, ami szóközökkel elválasztott számokat kér, és kiírja az átlagukat. Tehát a program egy futása ehhez hasonlóan nézhet ki:

```
Írjon be néhány számot szóközökkel elválasztva: 4 12 5
Az átlag  7.0
```

I/O: `print()`

Visszatérve a duplázós példához: jobb lenne az eredményt kicsit érthetőbben, valami kontextussal együtt prezentálni, mondjuk így:

```
A 21 duplája a 42.
```

A következő majdnem ezt teszi:

```
x = int(input('Adjon meg egy egész számot: '))
print ('A(z)', x, 'duplája a(z)', 2*x, '.')
```

Emlékeztető: `print()`-nek több argumentuma is lehet, ezeket szóközzel elválasztva nyomtatja ki. Sajnos ez az utolsó argumentumra is vonatkozik:

```
A(z) 21 duplája a(z) 42 .
```

I/O: `print()`

Egyszerű, ha nem is ideális megoldás erre a problémára a `print()` `sep` nevű ú.n. *keyword* argumentuma: ez határozza meg, hogy mi választja el az argumentumok nyomtatott képeit. Alapértelmezésben ez egy szóköz. Később, ld. itt, majd látni fogjuk az ideális megoldást: az `f-string`-eket.)

```
x = int(input('Adjon meg egy egész számot: '))
print ('A(z) ', x, ' duplája a(z) ', 2*x, '.', sep='')
```

A `sep` értéke tetszőleges `str`ing lehet.

I/O: `print()`

Az `end` a `print()` egy másik hasznos keyword argumentuma, ami azt határozza meg, mi nyomtatódik az összes argumenyum kinyomtatása után. (Alapértelmezésben ez az új sor, azaz newline, amit így lehet leírni: `'\n'`.) Például:

```
>>> for i in range(5): print(i,end='|')
...
0|1|2|3|4|
```

Ha meg akarunk szabadulni az utolsó `'|'`-tól, a következő a legrosszabb megoldás (de tudjunk róla, hogy ilyen is van): a ciklus végeztével töröljük az utolsó `'|'`-t egy backspace (`'\b'`) karakter nyomtatásával. `'\b'`, `'\n'`-hez hasonlóan ú.n. *backslash escape sequence*; ilyen még a `'\t'` (tabulátor) is. Ezeket, talán a `'\n'` kivételével, kerüljük el, ha lehet.

I/O: print()

Jobb megoldás a problémára:

```
>>> print('|'.join([str(i) for i in range(5)]))
0|1|2|3|4
```

Ha s string, l pedig stringek egy listája, akkor s.join(l) az l-beli stringek összefűzöttje s-ekkel elválasztva. Például:

```
>>> '<>'.join(['ez', 'kissé', 'furán', 'fest'])
'ez<>kissé<>furán<>fest'
```

I/O: file-ok olvasása

Tegyük fel, hogy a `data.txt` file ezt tartalmazza:

```
$ cat data.txt
one
two
three
very long
four
five
```

Python-ban így olvashatjuk és nyomtathatjuk ki `data.txt` sorait:

I/O: file-ok olvasása

```
>>> with open('data.txt') as in_file:
...     for line in in_file:
...         print(line.rstrip())
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'data.tx
```

`with` egy új *blokk*-ot nyit meg (ld. a sorvégi pontosvesszőt és a nagyobb indentálást a következő sorokban); mivel ezt

```
  open('data.txt') as in_file
```

követi, ez a konstrukció megnyitja (olvasásra) az aktuális directory-beli `data.txt` file-t és a `in_file` változóba tesz egy, a sorait tartalmazó iterálhatót (*iterable*).

I/O: file-ok olvasása

Az `.rstrip()` a sorok végén található szóköz és újsor karaktereket csípi le. (Próbáljuk ki, mi az eredmény, ha ezt elhagyjuk!)

Íme a programunk kicsit továbbfejlesztett változata:

```
>>> with open('data.txt') as in_file:
...     for i, line in enumerate(in_file,1):
...         print(str(i)+': '+line.rstrip())
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'data.tx
```

I/O: `open()`/`.close()`

Létezik file-ok megnyitására egy primitívebb mechanizmus is: az `open()` függvény. Itt az első file-olvasó programunk eredeti, és az `open()`-t használó változata:

```
with open('data.txt') as in_file:
    for line in in_file:
        print(line.rstrip())
```

```
in_file = open('data.txt')
for line in in_file:
    print(line.strip())
in_file.close()
```

Az első nem csak szebb, de bolondbiztosabb is, mert nem kell emlékeznünk arra, hogy a file-okat be is kell zárni. Márpedig ez fokozottan igaz, ha írunk beléjük, mert a bezárás elmulasztása könnyen adatvesztéssel járhat.

Úgyhogy, különleges esetektől eltekintve, használjuk a `with open()` szerkezetet.

I/O: file-ok írása

Írjuk egy `out.txt` nevű file-ba `data.txt` sorait fordított sorrendben:

```
>>> lines = []
>>> with open('data.txt') as in_file:
...     for line in in_file:
...         lines.append(line.rstrip())
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'data.txt'
>>> with open ('out.txt','wt') as out_file:
...     for line in lines[::-1]:
...         print(line,file=out_file)
...
>>> #ellenőrizzük az eredményt
>>> import os; print(os.popen('cat out.txt').read())
```

I/O: file-ok írása

Az történt, hogy megnyitottuk az `out.txt` nevű szövegfile-t (text mode) írásra (writing); ezeket jelenti `wt` az `open()` második argumentumában. Állhatna ott `rt` (read in text mode — ez az alapértelmezés), `rb` (read in binary mode) és `wb` (write in binary mode).

Másik újdonság itt a `print()` `file` nevű keyword argumentuma, aminek értéke lehet egy írásra megnyitott file; ennek hatására `print()` abba ír a képernyő (pontosabban a *standard output*) helyett.

Ugyanezt az eredményt tömörebb kóddal is elérhettük volna:

```
>>> with open('data.txt') as in_file:
...     with open ('out.txt','wt') as out_file:
...         for line in reversed(list(in_file)):
...             print(line.rstrip(),file=out_file)
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'data.txt'
```

Tárolók (tömbök)

listák, tuple-k és stringek

listákról már sokat tudunk, amit még nem, azt majd most megtanuljuk (vagy legalább megemlítjük). Azt is láttuk már, hogy a listákkal megtehető dolgok egy része tuple-kkal és/vagy stringekkel is megtehető, aminek az az oka, hogy ezek mind *sorozat*ok (leképezések a természetes számok egy kezdőszeletéből az összes Python objektumok, vagy a stringek esetében a karakterek, halmazába). Sorozatok még a range-ek (a range() függvény értékkészlete) is.

A listák és tuple-k közti legfőbb különbség az, hogy az utóbbiak nem módosíthatók (*immutable*), ahogy a stringek sem.

listák, tuple-k és stringek

```
>>> l = list(range(0,10,2)) ; l
[0, 2, 4, 6, 8]

>>> t = tuple(l) ; t  # tuple létrehozásának egyik módja
(0, 2, 4, 6, 8)

>>> l[3] = 5 ; l[3]
5

>>> t[3]
6

>>> t[3] = 5 #ez nem fog menni
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

listák, tuple-k és stringek

```
>>> s = 'abcdef ghijk' ; s
'abcdef ghijk'

>>> s[3]
'd'

>>> s[3] = 'q' #ez sem
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

De ezektől eltekintve minden működik tuple-kra és stringekre, amiket eddig tanultunk (leginkább a különféle indexelési módokat) listákról.

listák, tuple-k és stringek

És ezek is, amiket még nem:

```
>>> l + l, t + t
([0, 2, 4, 5, 8, 0, 2, 4, 5, 8], (0, 2, 4, 6, 8, 0, 2, 4, 6, 8))
>>> l*2, t*2
([0, 2, 4, 5, 8, 0, 2, 4, 5, 8], (0, 2, 4, 6, 8, 0, 2, 4, 6, 8))
>>> s + s, 2*s
('abcdef ghijkabcdef ghijk', 'abcdef ghijkabcdef ghijk')
```

Feladat
Miért működik az alábbi kódrészlet? Nem mond ez ellent annak, hogy a tuple-k nem módosíthatók?

```
>>> tup = ([0,1],[2]) ; tup[0][1] = 'a' ; tup
([0, 'a'], [2])
```

Segítség: próbáljuk ki itt!

`s[i]`	`s` `i`. eleme
`s[i:j]`	`s` egy szelete
`s[i:j:stride]`	`s` egy kiterjesztett szelete
`len(s)`	`s` hossza
`min(s)`	`s` legkisebb eleme
`max(s)`	`s` legnagyobb eleme
`sum(s [,initial])`	`s`-beli elemek összege; `str`ingekre nem alkalmazható – azokra használjunk `.join()`-t
`all(s)`	`True` csakkor ha `s` minden tagja `True`
`any(s)`	`True` csakkor ha `s`-nek van `True` tagja
`x in s`	`True` csakkor ha `x` tagja `s`-nek

Table 3: Műveletek és függvények sorozatokon

`s[i] = v`	tag hozzárendelés
`s[i:j] = v`	hozzárendelés szelethez
`s[i:j:stride] = v`	hozzárendelés kiterjesztett szelethez
`del s[i]`	tag törlése
`del s[i:j]`	szelet törlése
`del s[i:j:stride]`	kiterjesztett szelet törlése

Table 4: Műveletek `list`ákon

listák, tuple-k és stringek

Ezek a listák-ra ill. tuple-kre alkalmazható metódusok (IPython TAB-kiegészítésével előállítva[5]):

```
list.
  append()  count()   insert()  reverse()
  clear()   extend()  pop()     sort()
  copy()    index()   remove()

tuple.
  count() index()
```

[5] Más módjai annak, hogy megkapjunk ezek listáját: `dir(list)` vagy `dir(l)`, ahol `l` egy lista. Tehát például `dir([])` is működik. Ha a dokumentációjukkal együtt szeretnénk megkapni őket, akkor `help(list)` (vagy `help(l)`, ha `l` egy lista). Azonban az ezekkel a technikákkal (amelyek természetesen más típusokra is működnek) kapott metódusok közül hagyjuk figyelmen kívül azokat, amelyeknek aláhúzásjellel (_) kezdődik a nevük. Később (talán) majd meglátjuk, miért.

A listák néhány metódusa értelemszerűen hiányzik a tuple-k metódusai közül: például `l.sort()` "helyben" rendezi, azaz megváltoztatja az `l` listát[6]. Ugyanez igaz `.reverse()`-re is:

```
>>> l.reverse(); l
[8, 5, 4, 2, 0]
>>> l.sort(); l
[0, 2, 4, 5, 8]
```

ami nem azt jelenti, hogy egy tuple-t vagy egy stringet ne lehetne könnyen megfordítani:

[6] Van egy `sorted()` *függvény*, amely listákra, tuple-okra és stringekre is alkalmazható. Azonban bármilyen típusú bemenet esetén listát ad vissza. A `.reverse()` -nek is van függvény párja, a `reversed()`, de ez nem listát, hanem egy ú.n. *iterátor*t ad vissza

```
>>> t[::-1]
(8, 6, 4, 2, 0)
>>> s[::-1]
'kjihg fedcba'
```

de, ellentétben a `reverse()`-el, ezek természetesen nem változtatják meg magát a `tuple`-t vagy a `str`inget.[7]

[7] Írhatunk `t = t[::-1]`-et is, így a `t` értéke egy *új* `tuple` lesz, aminek a tagjai ugyanazok lesznek, mint az eredetié, csak fordított sorrendben. `t` értéke lett más, nem az eredeti értéke (a `tuple`) változott. A különbség megértéséhez próbáljuk ki ezt: `print(id(l)); l.reverse(); print(id(l))` és ezt: `print(id(t)); t = t[::-1]; print(id(t))`-t!

`l.append(obj)`, ahogy azt már láttuk, `obj` értékét az `l` listához adja, `l.extend(it)` pedig kiegészíti `l`-et az iterálható (list, tuple, string, ...) `it` tagjaival. A következő példa mutatja a kettő közti különbséget:

```
>>> l
[0, 2, 4, 5, 8]
>>> l.append(['a','b','c']) ;l
[0, 2, 4, 5, 8, ['a', 'b', 'c']]
>>> l.extend(['a','b','c']) ;l
[0, 2, 4, 5, 8, ['a', 'b', 'c'], 'a', 'b', 'c']
```

`.extend()` `+`-tól (összefűzés) is különbözik két dologban:

- `l.extend(it)` megváltoztatja `l`-et (ezért nincs ilyen metódusa `tuple`-nek és `str`ingnek), szemben `+`-al, ami új `list`át ad vissza.
- `.extend()` argumentuma tetszőleges iterálható lehet, míg `+` argumentumai csak azonos típusúak lehetnek:

```
>>> l = l[:5] ; l.extend('def') ; l
[0, 2, 4, 5, 8, 'd', 'e', 'f']
```

de

```
>>> l + 'def'     #nem fog menni
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "str") to list
```

- Az `.index()` metódus az argumentuma első előfordulásának indexét adja vissza abban a sorozatban, amire meghívtuk:

  ```
  >>> [0,1,'a',1,0].index(1)
  1
  ```

  Ha nem fordul elő benne, akkor ez a hívás `ValueError` hibát dob.

- A `.count()` metódus az argumentuma előfordulásainak számát adja vissza:

  ```
  >>> [0,1,'a',1,0].count(1)
  2
  ```

tuple-k

tuple-k tagjaihoz hozzá tudunk férni indexeléssel (mondjuk `t[2]`), de gyakrabban használjuk a változó-kicsomagolást (ld. itt!):

```
>>> t
(0, 2, 4, 6, 8)
>>> a, _, c, _, e = t
>>> c, a
(4, 0)
```

(A változó-kicsomagolás más sorozatokra is működik, de, legalábbis ebben az egyszerű formájában, kevésbé használatos.)

Az aláhúzás karakter itt (is) azt jelzi, hogy nem érdekel a megfelelő érték (most pl. nem akarjuk egy változóhoz rendelni).

tuple-k

A változó-kicsomagolás egyik gyakori felhasználási módja az, amikor egy tuple-kből álló iterálhatón megyünk végig. Láttunk már ilyet az enumerate() kapcsán. Itt egy másik példa.

Legyen három listánk: az első áruk neveit tartalmazza, a második és harmadik a megfelelő egységárakat ill. raktárkészleteket. Feladatunk egy (áru, összérték) párokból álló lista előállítása:

```
>>> goods = ['ball', 'table', 'racket', 'net']

>>> amounts = [570, 3, 12, 17]

>>> uprices = [0.13, 2000, 185, 23]

>>> [(good, a*up) for good, a, up in
... zip(goods, amounts, uprices)]
[('ball', 74.10000000000001), ('table', 6000), ('racket', 2220),
```

Újdonság itt a `zip()` függvény, ami a következőt csinálja: *n* db. iterálhatóval meghívva a közülük legrövidebbel azonos hosszúságú, olyan *n*-`tuple`-kből álló iterálhatót ad vissza, amelyeknek *i*. eleme az *i*. argumentum megfelelő tagja.

Például:

```
>>> list(zip([1, 2, 3, 4], ['a', 'b', 'c']))
[(1, 'a'), (2, 'b'), (3, 'c')]
```

Ez nagyon hasznos amikor több iterálható fölött párhuzamosan akarunk iterálni.

tuple-k

tuple-k úgy írhatók, mint a listák, csak szögletes helyett síma zárójelekkel határolva:

```
>>> type((1,2,3))
<class 'tuple'>
```

A vessző a lényeg, nem a zárójel:

```
>>> x = 1,2,3 #ahogy ezt már korábban is láttuk
>>> x, type(x)
((1, 2, 3), <class 'tuple'>)
```

"Elfajuló eset": egyelemű tuple. Valahogy jeleznünk kell, hogy tuple-re gondolunk (nem az egyetlen tagjára), ezért vesszőt kell írni az egyetlen tag után. (Bármely kifejezés zárójelbe tehető, az itt nem segítene):

```
>>> (1) == 1
True
>>> type((1)), type((1,))
(<class 'int'>, <class 'tuple'>)
```

változó-kicsomagolás

Még valami a változó-kicsomagolásról: mi van, ha egy értékadásnál nem tudjuk előre a jobb oldalon lévő `tuple`[8] hosszát? Kell tudnunk jelezni, hogy van ott egy változó, ami fogadja az összes olyan tagot, ami nem tartozik más változókhoz. És az "összes tag" csak "valamilyen gyűjtemény a tagokból" lehet. Erre a célra szolgál a a változó neve elé írandó `*`; és a "gyűjtemény" egy lista lesz.

```
>>> u,_,v,*w = list(range(10)) ; u,w
(0, [3, 4, 5, 6, 7, 8, 9])

>>> _,_,v,*w,u = tuple(range(10)) ; u,w
(9, [3, 4, 5, 6, 7, 8])

>>> u,_,v,*w = 'mi megy hova'; u,w
('m', ['m', 'e', 'g', 'y', ' ', 'h', 'o', 'v', 'a'])
```

[8] vagy `lista` vagy `string`

stringet négy különböző módon lehet megadni:

```
>>> 'ab' == "ab" == """ab""" == '''ab'''
True
```

Mindegyik jó valamire:

```
>>> print("No I don't") #símán írhatunk bele '-t
No I don't
>>> print('"Yes," he said') #símán írhatunk bele "-t
"Yes," he said
>>> #ez nem menne:
>>> #print("Túl hosszú, nem
>>> #fér el egy sorban.")
```

```
>>> print("""Túl hosszú, nem
... fér el egy sorban.""")  #hasznos többsoros string-ekhez
Túl hosszú, nem
fér el egy sorban.
>>> #de nem nélkülözhetetlen
>>> print("Túl hosszú, nem \
... fér el egy sorban.")
Túl hosszú, nem fér el egy sorban.
>>> print("Túl hosszú, nem \nfér el egy sorban.")
Túl hosszú, nem
fér el egy sorban.
```

str metódusok

```
str.
  capitalize()   encode()       format()       isalpha()
  casefold()     endswith()     format_map()   isascii()
  center()       expandtabs()   index()        isdecimal()
  count()        find()         isalnum()      isdigit()

  isidentifier() isspace()      ljust()        partition()
  islower()      istitle()      lower()        removeprefix()
< isnumeric()    isupper()      lstrip()       removesuffix()
  isprintable()  join()         maketrans()    replace()

  rfind()        rsplit()       startswith()   translate()
  rindex()       rstrip()       strip()        upper()
< rjust()        split()        swapcase()     zfill()
  rpartition()   splitlines()   title()
```

`str` metódusok

Ezek némelyikével már találkoztunk: pl. `.join()`-nal, `.rstrip()`-pel és `.split()`-tel a 3. (I/O) szakaszban. További hasznos metódusok:

- `.replace()` egy részstring (alapértelmezésben) összes előfordulását kicseréli egy máskra:

```
>>> s = "you think you can do it"
>>> s.replace("you","we")
'we think we can do it'
>>> s.replace("you","we",1)
'we think you can do it'
```

  Bonyolultabb kicserélésekhez *reguláris kifejezéseket* kell használni.

str metódusok

- `.split()` argumentumok nélkül a `str`ingbeli "szavak" listáját adja vissza:

  ```
  >>> s.split()
  ['you', 'think', 'you', 'can', 'do', 'it']
  ```

  De egy `str`ing argumentummal azt tekinti a "szavak" határának:

  ```
  >>> s.split("ou")
  ['y', ' think y', ' can do it']
  >>> [w.strip() for w in s.split("ou")]
  ['y', 'think y', 'can do it']
  ```

- `.strip()`, amit az utolsó példában használtunk, az `.rstrip()` szimmetrikus változata: argumentum nélkül a `str`ing két végén található szóköz és újsor karaktereket csípi le és az így kapott új `str`inget adja vissza.

`str` konverziók

A következő *függvény*ekkel már találkoztunk:

- `str()` az argumentumaként megadott objektum `str`ing reprezentációját adja vissza (ha van neki)

```
>>> 2**3+1
9
>>> str(2**3)+str(1)
'81'
```

- `int()` és `float()` a `str()` megfelelő megszorításának inverzei: ezekkel lehet egy `int` vagy egy `float` `str`ing reprezentációját (nyomtatott képét) `int`-é vagy `float`-tá konvertálni.[9]

```
>>> int("2")**int("3")+int("1")
9
>>> float("1.4142135623730951")**2
2.0000000000000004
```

[9] Más alkalmazásuk is van.

str konverziók

Ha int() vagy float() nem képes a string argumentumát int-é vagy float-tá konvertálni, akkor ValueError hibát dob:

```
>>> int("kilenc")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'kilenc'
```

Majd később látni fogjuk, hogyan lehet az ilyen hibákat kezelni.
Általában: fontos, hogy képesek legyünk különböző adattípusokat (mint int vagy float) stringekké konvertálni és viszont, mert a szöveg nagyon elterjedt kommunikációs formátum, pl. a web nyelve (HTML) és protokollja (HTTP) is ilyen.

f-stringek

Eddig ha valamit szépen akartunk kinyomtatni, a megoldás szinte mindig valami gányolás volt. Megfigyelés: a szép outputok mindig konstans és változó részekből állnak. Pl.:

```
A válasz: 42
```

ami a programunkban így nézhet ki:

```
print('A válasz: ', answer)
```

Más példa: Tegyük fel, hogy a következő "adatbázist", vagy egy abból származó adathalmazt akarunk valahogy prezentálni:

```
>>> db = [
...    ('ball', 570, 0.13),
...    ('table', 3, 2000),
...    ('racket', 12, 185),
...    ('net', 17, 23)
... ]
```

f-stringek

Ahogy például korábban tettük:

```
>>> [(good, a*up) for good, a, up in db]
[('ball', 74.10000000000001), ('table', 6000), ('racket', 2220),
```

Ezt csak mi értjük, mert mi látjuk a programot, ami előállította. Ahhoz, hogy másnak is hasznos legyen, minimum meg kell valahogy címkézni az adatokat, mondjuk így:

```
Name: ball
Total price: 74.10000000000001
Name: table
Total price: 6000
Name: racket
Total price: 2220
Name: net
Total price: 391
```

f-stringek

Itt is minden sorban van egy konstans string, mint pl. `Name:`, és egy változó szám (string reprezentációja), mint pl. 6000.

Ilyet az eddigi eszközeinkkel is könnyen tudunk csinálni, de az ideális megoldás az f-stringek (*formatted string literals*): ezek konstans stringek "lukakkal" (*replacement fields*), amik bajusz-zárójelek közé írt, a nyomtatás során kiértékelődő Python kifejezések. Például:

```
>>> f'1+1 = {1+1}'
'1+1 = 2'
>>> f'Az adatbázis első sora {db[0]}'
"Az adatbázis első sora ('ball', 570, 0.13)"
```

Az `f` karakter jelzi, hogy ami utána következik, az nem közönséges string, mert ilyen lukak lehetnek benne.

f-stringek

Ezt használva már egészen elfogadható formában prezentálhatjuk az adatbázisunkat:

```
>>> for good, a, up in db:
...     print(f'Name: {good}\nTotal price: {a*up}')
...
Name: ball
Total price: 74.10000000000001
Name: table
Total price: 6000
Name: racket
Total price: 2220
Name: net
Total price: 391
```

f-stringek

Az f-stringekben azt is meg lehet mondani, hogy a változó részek milyen módon jelenjenek meg. Például ha nem akarunk ilyen sok tizedesjegyet, {a*up} helyett {a*up:.2f}-t írhatunk (a kettőspont utáni rész hivatalos neve *format specifier*). Ez biztosítja, hogy a szám float-ként, 2 tizedes pontossággal kerül a helyére:

```
>>> for good, a, up in db:
...     print(f'Name: {good}\nTotal price: {a*up:.2f}')
...
Name: ball
Total price: 74.10
Name: table
Total price: 6000.00
Name: racket
Total price: 2220.00
Name: net
Total price: 391.00
```

f-stringek

Előírhatjuk, hogy a "lukak" milyen szélesek legyenek, ami hasznos, ha táblázatos formában akarunk nyomtatni:

```
>>> def doit():
...     print(f'{"Name":20}Total price')
...     for good, a, up in db:
...         print(f'{good:20}{a*up:10.2f}')
...
>>> doit()
Name                Total price
ball                     74.10
table                  6000.00
racket                 2220.00
net                     391.00
```

f-stringek

Itt az az újdonság, hogy `"Name":20` és `good:20` miatt `"Name"` és good 20 karakter szélességű oszlopokba kerül, és `a*up:10.2f` miatt a*up 10 karakter széles oszlopba íródik, jobbra igazítva, mert ez egy numerikus mező. Balra `a*up:<10.2f`-el, középre `a*up:^10.2f`-el tudnánk igazítani.

Néhány további példa:

```
>>> ans = 42
>>> print(f'|{ans:7d}|'); print('|1234567|') # 'd': 'decimal'
|     42|
|1234567|
>>> print(f'|{ans:<7d}|'); print('|1234567|')
|42     |
|1234567|
>>> print(f'|{ans:^7d}|'); print('|1234567|')
|  42   |
|1234567|
```

f-stringek

```
>>> print(f'|{ans:07d}|'); print('|1234567|') #feltöltés '0'-kkal
|0000042|
|1234567|
>>> print(f'|{ans:7b}|'); print('|1234567|') # 'b': 'binary'
| 101010|
|1234567|
>>> print(f'|{ans:7.2f}|'); print('|1234567|')
|  42.00|
|1234567|
>>> print(f'|{ans:07.2f}|'); print('|1234567|')
|0042.00|
|1234567|
```

Itt található az f-stringek hivatalos dokumentációja.

`dict` (szótár)

`dict` abban különbözik a `list`ától, hogy nem természetes számok, hanem (majdnem) tetszőleges Python objektumokkal (*kulcs*okkal) lehet megcímezni a benne tárolt adatokat. Például a korábbi "árukészlet" adatbázist sokkal természetesebb lenne szótárban tárolni, és az árukhoz tartozó adatokat az áruk nevével megcímezni (ld. alább).

```
>>> d = dict()
>>> type(d)
<class 'dict'>
>>> d
{}
>>> d['one']=1 ; d['two']=2; d
{'one': 1, 'two': 2}
>>> d['two']
2
>>> d.get('two')
2
```

dict (szótár)

dict létrehozása, feltöltése és lekérdezése:

```
>>> 'two' in d, 'three' in d
(True, False)
>>> d['three']  #nem fog menni
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'three'
>>> None == d.get('three')
True
>>> d.get('three',"can't find it")
"can't find it"
>>> d.get('two',"can't find it")
2
```

Amint itt látható, a .get() metódusnak van egy második, opcionális argumentuma (alapértelmezésben None), amit akkor ad vissza, ha a *kulcs* (az első argumentuma) nem szerepel a szótárban

`dict` (szótár)

A szótárak metódusai:

`clear()`	`get()`	`pop()`	`update()`
`copy()`	`items()`	`popitem()`	`values()`
`fromkeys()`	`keys()`	`setdefault()`	

Például `items()` és `values()` segítségével iterálhatunk a `dict` fölött:

dict (szótár)

```
>>> #iterálás a kulcsokon
>>> for key in d: print(key)
...
one
two
>>> #iterálás az értékeken
>>> for val in d.values(): print(val)
...
1
2
>>> #iterálás a kulcs-érték párokon
>>> for key,val in d.items(): print(key, val)
...
one 1
two 2
```

dict (szótár)

Ahogy listákat és stringeket, úgy dicteket is megadhatunk közvetlenül: például a fenti d-t

```
d = dict()
d['one']=1 ; d['two']=2
```

helyett megadhattuk volna `d = {'one': 1, 'two': 2}`-vel is.

Most már jöhet fenti árukészlet adatbázis szótárral:

```
>>> db
[('ball', 570, 0.13), ('table', 3, 2000), ('racket', 12, 185), (
>>> total_values = {good: a*up for good, a, up in db}
>>> total_values
{'ball': 74.10000000000001, 'table': 6000, 'racket': 2220, 'net'
>>> total_values['ball']
74.10000000000001
```

`dict` (szótár)

A `total_values` `dict` szótárgenerálással készült, ami a listageneráláshoz hasonlít, csak normális helyett bajusz-zárójelek határolják, és tetszőleges objektumok helyett "kulcs:érték" párokat gyűjt össze.

Más példa szótárgenerálásra:

```
>>> szavak = 'némely szavak hosszabbak mint mások'.split(); szavak
['némely', 'szavak', 'hosszabbak', 'mint', 'mások']
>>> hosszak = {szo:len(szo) for szo in szavak}; hosszak
{'némely': 6, 'szavak': 6, 'hosszabbak': 10, 'mint': 4, 'mások': 5}
>>> hosszak['mint']
4
```

Kivételkezelés

Egy program futása során előfordulhatnak olyan kivételes helyzetek, amelyekkel nem tud mit kezdeni. Például:

```
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

vagy

```
>>> 1 / int('kettő')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'kettő'
```

A hibát okozó adatok jöhetnek pl. egy felhasználótól.

Az ilyen hibák nem kell, hogy végzetesek legyenek. A `try: ... except: ...` konstrukció segítségével megmenthetjük a helyzetet.

Kivételkezelés

```
>>> try:
...     print(1/0)
... except:
...     print("Valami nem stimmel.")
...
Valami nem stimmel.
```

vagy

```
>>> try:
...     print(1 / int('kettő'))
... except:
...     print("Valami nem stimmel.")
...
Valami nem stimmel.
```

Kivételkezelés

Miközben persze:

```
>>> try:
...     print(1/2)
... except:
...     print("Valami nem stimmel.")
...
0.5
```

Ez nem oldja meg a problémát, csak a szőnyeg alá söpri. De adhatunk a felhasználónak (ha úgy gondoljuk, hogy ő a hiba forrása) esélyt a korrigálásra:

Kivételkezelés

```
def divide():
    divisor = int(input("Mi legyen az osztó? "))
    return 1 / divisor

try:
    print(divide())
except:
    print("Sajnos nem jó. Próbálja meg újra!")
    print(divide())
```

Ez jobb, csak a program (és így a használója) nem tudja, mi is volt a baj. `except` minden hibát elkap, tehát az sem biztos, hogy az input volt hibás. Az is lehet, hogy más okokból épp kifogytunk a memóriából, és az `MemoryError`-hoz vezetett.

Kivételkezelés

A hibák, mint `ValueError` (amiket most `try: ... except: ...` elnyom), valójában objektumok, amiknek függvényében dönthet a program a folytásról, így:

```
try:
    print(divide())
except ValueError:
    print("SZÁMot kérek, nem szöveget!")
    print(divide())
except ZeroDivisionError:
    print("Sajnos nem tudok 0-val osztani.")
    print(divide())
```

Kivételkezelés

Vagy akár így:

```
while True:
    try:
        print(divide())
        break
    except ValueError:
        print("SZÁMot kérek, nem szöveget!")
    except ZeroDivisionError:
        print("Sajnos nem tudok 0-val osztani.")
```

és akkor kedvére próbálkozhat a felhasználó.

Kivételkezelés

```
Mi legyen az osztó? nulla
SZÁMot kérek, nem szöveget!
Mi legyen az osztó? 0
Sajnos nem tudok 0-val osztani.
Mi legyen az osztó? 5
0.2
```

Ez sokkal jobb, mert informatívak a hibaüzenetek, és akkor is a megfelelő dolog történik, ha valami más, váratlan probléma merül fel: mivel azt nem "kezeljük", ezért pl. nem kínáljuk fel a lehetőséget a szám ismételt bevitelére (nagyon helyesen). De egy kicsit még szépíthetünk a nem várt hiba (nem) kezelésén is, így:

Kivételkezelés

```
while True:
    try:
        print(divide())
        break
    except ValueError:
        print("SZÁMot kérek, nem szöveget!")
    except ZeroDivisionError:
        print("Sajnos nem tudok 0-val osztani.")
    except:
        print("\nNem tudom mi történt, sajnos fel kell adjam!")
        raise
```

Kivételkezelés

```
Mi legyen az osztó? nulla
SZÁMot kérek, nem szöveget!
Mi legyen az osztó?       #itt Ctrl-d-t (EOF) nyomtam
Nem tudom mi történt, sajnos fel kell adjam!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/tmp/mc.py", line 67, in <module>
  File "/tmp/mc.py", line 62, in divide
EOFError
>>>
```

Itt az utolsó `except`-re akkor kerül a sor, ha a hiba nem `ValueError` vagy `ZeroDivisionError`; ilyenkor viszont `raise` ugyanazt a hibát dobja, ami miatt ide kerültünk. Ez azért jó, mert így esélyt adunk a felhasználónak (vagy a programnak, aminek kis részlete a mienk), hogy a hiba típusától függően cselekedjen.

Kivételkezelés

Más példa: színészek népszerűségét akarjuk felmérni úgy, hogy mindenki szavazhat a kedvencére.

- A szavazás mindenkori állapotát `dict`-ben tároljuk:
  - a kulcsok a nevek
  - az értékek az eddig a névre beérkezett szavazatok száma.
- Mivel egyrészt nincs katalógusunk az összes létező színészről, másrészt a túlnyomó többségükre nem érkezik szavazat, nem tudjuk és nem is akarjuk az összeset előre betenni a szótárba 0 értékkel.
- Így viszont muszáj valami speciálisat tennünk, amikor valakire először érkezik szavazat.

Kivételkezelés

```
>>> votes = {} #üres dict

>>> def vote(name):
...     if name in votes:
...         votes[name]+=1
...     else:
...         votes[name]=1
...
>>> vote('Brad Pitt'); vote('Julia Roberts'); vote('Brad Pitt')
>>> votes
{'Brad Pitt': 2, 'Julia Roberts': 1}
```

Más megoldás: nem ellenőrizzük, hogy szerepel-e már a név a szótárban.
Ha nem, kezeljük az emiatt előálló (`KeyError`) hibát:

Kivételkezelés

```
>>> votes = {}

>>> def vote(name):
...     try:
...         votes[name]+=1
...     except KeyError:
...         votes[name]=1
...
>>> for n in ['Brad Pitt', 'Julia Roberts', 'Brad Pitt', \
... 'Julia Roberts', 'Chris Pratt', 'Julia Roberts', \
... 'Michelle Yeoh']:
...     vote(n)
...
```

Kivételkezelés

```
>>> for name, n in votes.items():
...     print(f'{name:20} {n:5d}')
...
Brad Pitt                2
Julia Roberts            3
Chris Pratt              1
Michelle Yeoh            1
```

Ami most jön annak semmi köze kivételkezeléshez.

Ha meg akarjuk nézni, ki a két legnépszerűbb színész, a `dict .items()` metódusát használhatjuk, ami az összes kulcs-érték párt adja vissza. Ezt rendezhetjük (a szavazatok száma szerint csökkenő sorrendben) a `sorted()` függvény segítségével:

Kivételkezelés

```
>>> sorted(votes.items(), reverse = True, key=lambda kv: kv[1])[:2]
[('Julia Roberts', 3), ('Brad Pitt', 2)]
```

- `sorted()` azért rendez csökkenő sorrendben, mert a `reverse` keyword argumentum `True`.
- `sorted()` `key` nevű keyword argumentuma (aminek semmi köze a `dict`-beli kulcsokhoz!) egy függvény lehet, ami egy kulcs-érték párból kiszámolja azt az értéket, aminek alapján rendezni akarunk. Nálunk ez az érték, vagyis a szavazatok száma.

Ha csak a legnépszerűbb színész érdekel, használhatjuk a `max()` függvényt is, aminek szintén adhatunk `key` keyword argumentumot, hasonló céllal:

```
>>> max(votes.items(), key=lambda kv: kv[1])
('Julia Roberts', 3)
```

Kivételkezelés: további részletek

- Több hibát is kezelhetünk egy `except` ágban: ilyenkor ezeket egy `tuple` komponenseiként kell írni. Például:

  ```
  except (ValueError, ZeroDivisionError):
  ```

- A `try: ... except: ...` blokknak lehet egy `else:` ága is. Ez akkor hajtódik végre, ha a `try:` sikeresen (hiba nélkül) hajtódott végre. A különbség a következő kettő között

```
try:
    tedd_ezt
    tedd_ezt_ha_minden_OK
except:
    tedd_ezt_ha_gond_volt
```

```
try:
    tedd_ezt
except:
    tedd_ezt_ha_gond_volt
else:
    tedd_ezt_ha_minden_OK
```

akkor bukik elő, ha `tedd_ezt_ha_minden_OK` hibát dob.

Kivételkezelés: további részletek

- Használhatunk `finally:` ágat is: ez mindenképpen végrehajtódik, akár dobott hibát valamelyik `try` ágbeli utasítás, akár nem.

```
while True:
    try:
        print(divide())
        break
    except ZeroDivisionError:
        print("Sajnos nem tudok 0-val osztani.")
    finally:
        print("Remélem sikerült! Én nem tudom, mi történt.")
    print("Próbáljuk meg újra!")
```

`"Próbáljuk meg újra!"` csak akkor nyomtatódik ki, ha `divide()` hibát dobott, mert amúgy a `break` miatt kikerülünk a ciklusból, de a `"Remélem sikerült!..."` ettől függetlenül igen, mert a "mindenképpen végrehajtódik" szó szerint értendő.

Kivételkezelés: `assert`

Legalább két olyan szituáció van, amiben érdemes nem általunk definiált kivételt dobni (*raise an exception*):

- hibakezelés végén (ezt már láttuk)
- amikor ellenőrzünk egy olyan feltételt, amelynek fenn kell állnia ahhoz, hogy a programunk tovább tudjon futni (tipikusan a fejlesztés során, hibakeresést megkönnyítendő).

Ez utóbbi célra szolgál az `assert`: kötelező argumentuma egy általánosított boole érték amelynek teljesülése esetén a futás úgy folytatódik, mintha az `assert` ott se lenne. Ellenkező esetben viszont egy `AssertionError` kivételt dob.

Kivételkezelés: assert

```
>>> for n in range(10):
...     assert n % 2 == 0
...     print("Minden rendben.")
...
Minden rendben.
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
AssertionError
>>> #kezelhetnénk is, de minek; inkább:

>>> for n in range(10):
...     assert n % 2 == 0, f"Baj van: {n} nem páros"
...     print("Minden rendben.")
...
Minden rendben.
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
AssertionError: Baj van: 1 nem páros
```

Kivételkezelés: `assert`

Azért nem akarjuk kezelni, mert nem az a cél, hogy tovább fusson a program, hanem az, hogy kiderüljön: probléma van. Akkor viszont az `assert` második, opcionális argumentumát (*assertion message*) egyszerűbb használni, mint egy `try: ... except: ...` konstrukciót, aminek az `except AssertionError:` ágában is csak valami informatív szöveget nyomtatnánk ki.

`assert`-ekre nem érdemes támaszkodni kész programban, mert ki is kapcsolható az ellenőrzésük.

Haladó listagenerálás

Tudjuk, hogy ha `l` `lista`, akkor

```
result = [expr for i in l]
```

ekvivalens a következővel:

```
result = []
for i in l:
    result.append(expr)
```

és

```
result = [expr for x in l if c]
```

ekvivalens ezzel:

```
result = []
for i in l:
    if c:
        result.append(expr)
```

Haladó listagenerálás

Például

```
>>> [i**2 for i in range(20) if i%2 == 1]
[1, 9, 25, 49, 81, 121, 169, 225, 289, 361]
```

Haladó listagenerálás

De általánosabban:

```
[expr for i1 in l1 if c1
      for i2 in l2 if c2
      ...
      for iN in lN if cN ]
```

ezzel ekvivalens:

```
result = []
for i1 in l1:
    if c1:
        for i2 in l2:
            if c2:
                ...
                for iN in lN:
                    if cN:
                        result.append(expr)
```

Haladó listagenerálás

Például

```
>>> [(i,j) for i in [1,2,3] for j in ['a','b']]
[(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b'), (3, 'a'), (3, 'b')]
```

mert ez “balról jobbra” olvasandó, szemben az egymásba ágyazott listagenerálással:

```
>>> [[(i,j) for i in [1,2,3]] for j in ['a','b']]
[[(1, 'a'), (2, 'a'), (3, 'a')], [(1, 'b'), (2, 'b'), (3, 'b')]]
```

ami “kívülről befelé”.

Haladó listagenerálás

Itt egy másik példa, ami mutatja, hogy egy listagenerálásban, mint pl. ebben:

```
[(i,j) for i in [1,2,3] for j in ['a','b']]
```

a "belső" ciklusban (`for j in ...`) rendelkezésünkre áll a "külső ciklus" (`for i in ...`) által létrehozott lokális változó:

```
>>> [j for i in range(3)
...     for j in [range(3),range(3,6),range(6,8)][i]]
[0, 1, 2, 3, 4, 5, 6, 7]
```

Ennek érdemes a lényegét egy függvénybe kiabsztrahálni:

Haladó listagenerálás

Feladat
Írjon egy listagenerálást használó `concatenate()` nevű függvényt, amelynek argumentuma listák egy listája lesz, és ezek egymás után fűzöttjét adja vissza. Például:

```
>>> concatenate(
...   [list(range(3)),list(range(3,6)),list(range(6,8))])
[0, 1, 2, 3, 4, 5, 6, 7]
```

Az eredeti példa egy feltétellel:

```
>>> [(i,j) for i in [1,2,3] if i%2 == 1 for j in ['a','b']]
[(1, 'a'), (1, 'b'), (3, 'a'), (3, 'b')]
```

A "konkatenáló" példa több feltétellel:

```
>>> [j for i in range(3) if i%2 == 1
...  for j in [range(3), range(3,10), range(11,15)][i]
...  if j%2==0]
[4, 6, 8]
```

Függvények

Függvények

Emlékeztető:

```
def fun(par_1,par_2,...):
    statement_1
    statement_2
    ...
```

Ha egy így definiált függvényt hívunk a következő módon:

```
fun(arg_1,arg_2,...)
```

akkor az argumentumok (`arg_1`, `arg_2`,...) kiértékelődnek (tehát ha pl. `arg_1` egy másik függvényhívás, akkor az a függvény `fun` előtt hívódik meg), és aztán a függvény testében levő `statement_1`, `statement_2`, ... hajtódnak végre / értékelődnek ki, de úgy, hogy `par_1` értéke `arg_1`, `par_2` értéke `arg_2`, és így tovább. `par_1`, ... a függvény *paraméterei*. A függvény testében ezt *lokális változók*, tehát ha van velük azonos nevű változó a programban, azt *leárnyékolják* (*shadow*), azaz annak az értéke nem látható vagy változtatható meg a függvényben.

Függvények: lokális változók

A paramétereken kívül lokális változó lesz minden más olyan változó is, aminek értéket adunk a függvény testében. Például:

```
>>> a = 1 ; b = 2
>>> def some_fun(a):
...     b = a
...     print(f'A függvény testében\
...  a={a} és b={b} az értékadás után')
...
>>> some_fun(42)
A függvény testében a=42 és b=42 az értékadás után
>>> a,b
(1, 2)
```

Megjegyzés
Elég új Pythonban `f-string`-ben tetszőleges `expr` kifejezésre `expr={expr}` helyett írhatunk `{expr=}`-t.

Függvények: lokális változók

`b` globális értékéhez hozzáférünk a függvény testében, de csak ha nem hozunk létre ilyen névvel lokális változót:

```
>>> a = 1 ; b = 2
>>> def some_fun(a):
...     print(f'A függvény testében {a=} és {b=}')
...
>>> some_fun(42)
A függvény testében a=42 és b=2
>>> a,b
(1, 2)
```

Függvények: lokális változók

De a következő nem működik

```
>>> a = 1 ; b = 2
>>> def some_fun(a):
...     print(f'A függvény testében {b=} az értékadás előtt')
...     b = a
...     print(f'A függvény testében {a=} és {b=} az értékadás után')
...
>>> some_fun(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in some_fun
UnboundLocalError: cannot access local variable 'b' where it is not ass
```

mert még ha később is történik az az értékadás, a Python tudja, hogy létrehoztunk `b` nevű lokális változót (és használtuk mielőtt értéket adtunk volna neki).

Függvények: lokális változók

Sőt:

```
>>> a = 1 ; b = 2
>>> def some_fun(a):
...     print(f'A függvény testében {b=} az értékadás előtt')
...     if False:
...         b = a
...         print(f'A függvény testében {a=} és {b=} az értékadás után'
...
>>> some_fun(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in some_fun
UnboundLocalError: cannot access local variable 'b' where it is not ass
```

Hiába garantáltan nem hajtódik végre `b = a`, szerepel a függvény testében, ezért `b` lokális lesz.

Függvények: lokális változók

Ha valamilyen oknál fogva mégis meg akarjuk változtatni a globális `b` változó értékét a függvényben, a következőt tehetjük:

```
>>> a = 1 ; b = 2
>>> def some_fun(a):
...     global b
...     b = a
...     print(f'A függvény testében {a=} és {b=} az értékadás után')
...
>>> some_fun(42)
A függvény testében a=42 és b=42 az értékadás után
>>> a,b
(1, 42)
```

De ne tegyük. (Bár nagyon ritkán ugyan, de van amikor ez jó ötlet.)

Függvények: nemkívánt mellékhatások

A globális változók értékét (általában véletlenül) másképp is meg lehet változtatni, ahogyan azt a következő példa mutatja. Tegyük fel, hogy egy olyan függvényt akarunk definiálni, amely egy számlistát a "buborékrendezés" algoritmus segítségével rendez. Ennek az algoritmusnak az a lényege, hogy valahányszor egy számpárt rossz sorrendben találunk, felcseréljük őket.

Függvények: nemkívánt mellékhatások

```
>>> def bubble(lst):    #rossz
...     for i in range(len(lst)):
...         for j in range(len(lst[i+1:])):
...             jj = i+1+j
...             if lst[jj]<lst[i]:
...                 lst[jj], lst[i] = lst[i], lst[jj]
...     return lst
...
>>> import random
>>> l = [random.randint(0,100) for _ in range(10)]
>>> l
[82, 21, 69, 61, 79, 34, 100, 79, 31, 75]
>>> bubble(l)
[21, 31, 34, 61, 69, 75, 79, 79, 82, 100]
```

Ez jónak tűnik, de van vele egy kis gond:

```
>>> l
[21, 31, 34, 61, 69, 75, 79, 79, 82, 100]
```

Függvények: nemkívánt mellékhatások

A függvénynek nem kellett volna megváltoztatnia az argumentumát.[10] Itt a probléma egyszerűbb kontextusban:

```
>>> def side_effect(a):
...     a=42
...
>>> b = 0; side_effect(b); b
0
>>> #eddig minden rendben
>>> def side_effect(a):
...     a[0]=42
...
>>> b = [0]; side_effect(b); b
[42]
```

[10] Ha egy függvény vagy metódus célja, hogy mellékhatása legyen, például megváltoztassa az argumentumait, akkor az a szokás, hogy nem ad vissza értéket.

Függvények: nemkívánt mellékhatások

Itt nem az történt, hogy b-nek új értéket adtunk (ez nem is történhetett volna meg, mert nincs `global b` deklaráció a függvény definíciójában), hanem az, hogy a régi értéke változott meg.

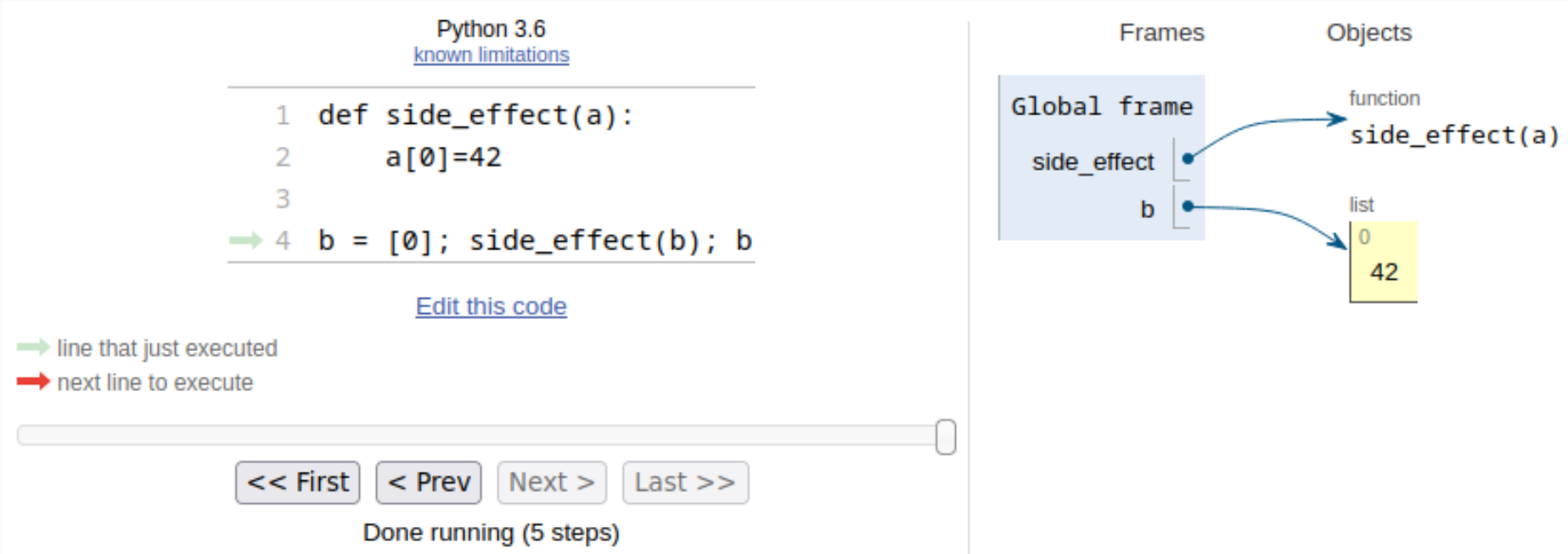


Figure 1: A memória állapota (Pythontutor)

Függvények: nemkívánt mellékhatások

Az ok az, hogy az ilyen összetett típusoknál, mint a `lista` (és minden más tömb, a `string`et kivéve) a változó értéke nem maga az objektum, hanem annak memóriabeli címe.

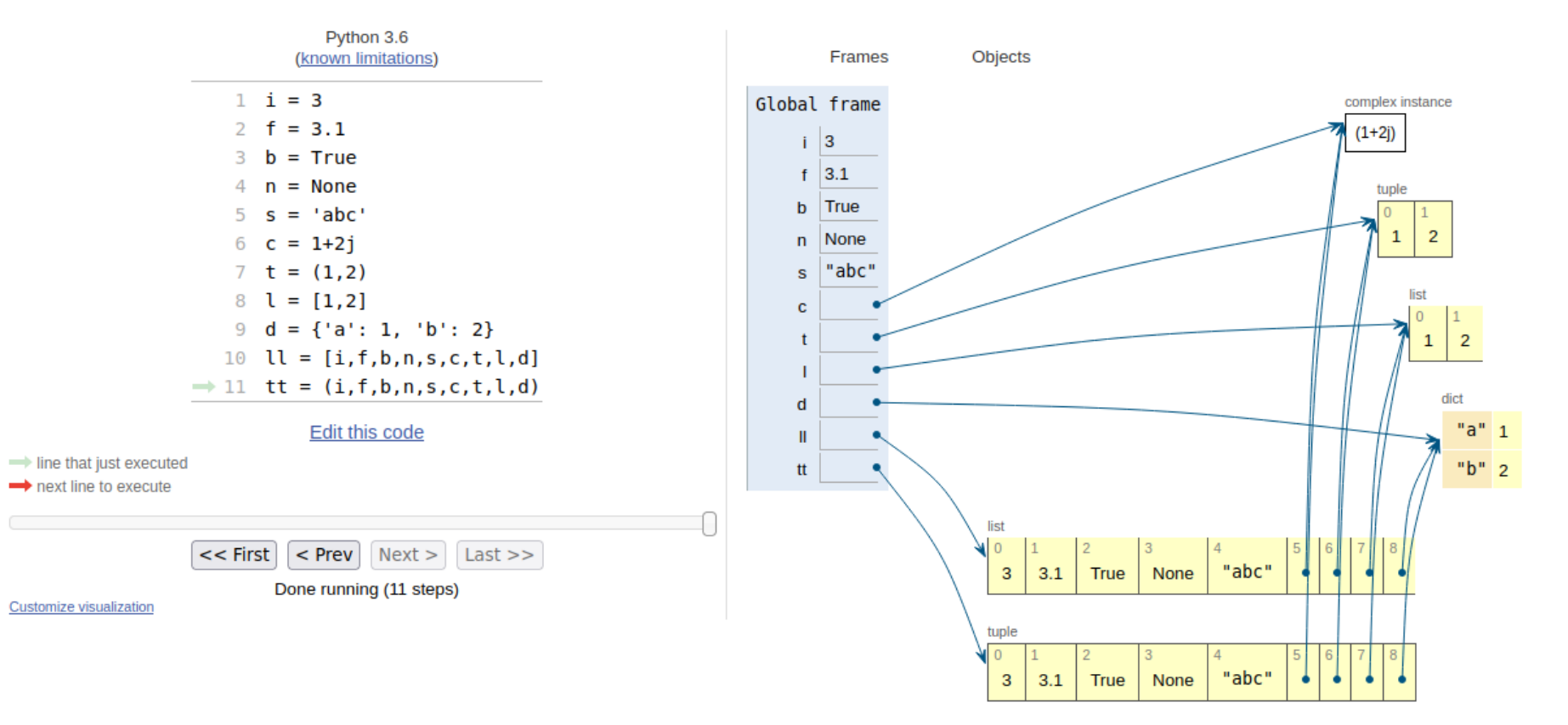


Figure 2: A memória állapota (Pythontutor)

Függvények: nemkívánt mellékhatások

És egy `list`a, mivel módosítható, megváltozhat anélkül, hogy a címe megváltozna. Ez történt `b` értékével a `side_effect()`-es példában.

A megoldás: másolatot kell készíteni az összetett objektumról, és azt módosítani:

```
>>> def side_effect(a):
...     a = a[:] #vagy a.copy(); vagy list(a)
...     a[0]=42
...
>>> b = [0]; side_effect(b); b
[0]
```

Itt a definíció második sorában új lokális változót hoztunk létre és ahhoz rendeltük a paraméter értékének egy másolatát.

Függvények: nemkívánt mellékhatások

De még ez az óvatosság sem mindig elég:

```
>>> def side_effect(a):
...     a = a[:]
...     a[0][0]=42
...
>>> b = [[0]]; side_effect(b); b
[[42]]
```

A probléma az, hogy `a[:]` és `a.copy()` egy ún. *shallow copy*-t hoz létre a listából. Ez új lista ugyan, de ha az eredeti tartalmazott egy listát, akkor valójában egy arra való hivatkozást tartalmazott, és ez a hivatkozás átmásolódik az új listába. Tehát ugyanaz a probléma fog felmerülni, csak más szinten. Amire itt szükségünk van, az egy *deep copy*, amely a hivatkozás másolása helyett (bármely szinten) létrehoz egy új listát (amely ismét annak a listának a deep copy-ja, amelyre a hivatkozás utalt), és *azt* teszi az újonnan létrehozott listába.

Függvények: nemkívánt mellékhatások

Valami ilyesmi

```
def deep_copy(l):
    return [deep_copy(i) for i in l] if isinstance(l, list) else l
```

megoldaná a problémát, legalábbis amíg a másolandó `list`ák legmélyén is csak `list`ák vannak az egyszerű értékeken kívül.

```
>>> def side_effect(a):
...     a = deep_copy(a)
...     a[0][0]=42
...
>>> b = [[0]]; side_effect(b); b
[[0]]
```

Függvények: nemkívánt mellékhatások

De az igazi megoldás, amire nincsenek ilyen megkötések, az a `copy` modulbeli `deepcopy()` függvény:

```
>>> import copy
>>> def side_effect(a):
...     a = copy.deepcopy(a)
...     a[0][0]=42
...
>>> b = [[0]]; side_effect(b); b
[[0]]
```

Ez minden összetett értékkel elbánik, nem csak a `list`ákkal.

A buborékrendezés példában erre nincs szükség, mert a rendezendő lista csak számokat tartalmaz.

Függvények: nemkívánt mellékhatások

```
>>> def bubble(lst):
...     lst = lst[:]
...     for i in range(len(lst)):
...         for j in range(len(lst[i+1:])):
...             jj = i+1+j
...             if lst[jj]<lst[i]:
...                 lst[jj], lst[i] = lst[i], lst[jj]
...     return lst
...
>>> l = [random.randint(0,100) for _ in range(10)]
>>> l
[90, 71, 62, 80, 43, 48, 39, 20, 7, 25]
>>> bubble(l)
[7, 20, 25, 39, 43, 48, 62, 71, 80, 90]
>>> l
[90, 71, 62, 80, 43, 48, 39, 20, 7, 25]
```

Függvények: nemkívánt mellékhatások

Nemkívánt mellékhatások nem csak függvényekben léphetnek fel:

```
>>> b = [[0]] ; c = b ; c[0][0] = 42; b
[[42]]
```

De a megoldás természetesen itt is ugyanaz:

```
>>> b = [[0]] ; c = copy.deepcopy(b) ; c[0][0] = 42; b
[[0]]
```

Függvények: keyword és opcionális argumentumok

A következő függvényhívásban

```
>>> f(1,2)
x=1, y=2
```

feltéve, hogy `f` így van definiálva:

```
def f(x,y):
    print(f'{x=}, {y=}')
```

Python a paraméterek pozíciójából tudja, hogy `1`-et `x`-hez, `2`-t `y`-hoz kell rendelni. Ezért ezeket szokás *pozicionális paraméterek*nek is nevezni. De *keyword argumentumok*kal explicit módon is előírhatjuk, hogy mely paraméterek mely értékeket kapják

```
>>> f(y=2,x=1)
x=1, y=2
```

Függvények: keyword és opcionális argumentumok

Megváltoztathatjuk a függvény definícióját úgy, hogy bizonyos argumentumokat csak keyword argumentumokként lehessen átadni. Például:

```
>>> def f(x,*,y):
...     print(f'{x=}, {y=}')
...
>>> f(1,2) #rossz
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() takes 1 positional argument but 2 were given
>>> f(1) #rossz
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() missing 1 required keyword-only argument: 'y'
>>> f(1,y=2) #végre...
x=1, y=2
```

Függvények: keyword és opcionális argumentumok

A függvény definíciójában megadhatunk alapértelmezett (*default*) értékeket a pozicionális paramétereknek:

```
>>> def f(x,y=3):
...     print(f'{x=}, {y=}')
...
>>> f(1,2)
x=1, y=2
>>> f(1)
x=1, y=3
```

és a csak keyword-ként használható (*keyword only*) paramétereknek is:

```
>>> def f(x,*,y=3):
...     print(f'{x=}, {y=}')
...
>>> f(1,y=2)
x=1, y=2
>>> f(1)
x=1, y=3
```

Függvények: keyword és opcionális argumentumok

De nem tehetünk alapértelmezett érték nélküli pozicionális paramétereket alapértelmezett értékkel ellátott paraméter után:

```
>>> def f(x,y=3,z):  #rossz
  File "<stdin>", line 1
    def f(x,y=3,z):  #rossz
                 ^
SyntaxError: non-default argument follows default argument
```

Ez logikus is, mert ha így kezdődik `f` definíciója:

```
def f(x,y=3,z):
```

és `f`-et két argumentummal hívjuk (és miért ne, amikor a három paraméter közül egynek van alapértelmezett értéke), milyen alapon dönthetné el Python, hogy a második argumentum melyik paraméterhez rendelődjön: `y`-hoz, ami a megfelelő pozícióban van, vagy `z`-hez, ami teljesen ad hoc választás lenne, de legalább minden paraméter kapna értéket?

Függvények: keyword és opcionális argumentumok

Ha keverni akarjuk az alapértelmezett és egyéb paramétereket, használjunk keyword only argumentumokat:

```
>>> def f(x,*,y=3,z):
...     print(f'{x=}, {y=}, {z=}')
...
>>> f(1,z=2)
x=1, y=3, z=2
```

Függvények: változó számú argumentum

Definiálhatunk változó számú argumentumszámú függvényeket is: ha az egyik paraméter neve *-al kezdődik[11], akkor abba kerül (`tuple`-ként) az összes *maradék* argumentum. (Azért maradék, mert közülük némelyik esetleg korábbi, pozicionális paraméterekhez rendelődött.)

Például tegyük fel, hogy előre nem ismert számú szám átlagát akarjuk kiszámolni.

```
>>> def avg(*nums):
...     return(sum(nums)/len(nums))
...
>>> avg(2,3)
2.5
>>> avg(2,3,5,9,6)
5.0
```

[11] V.ö. a változó-kicsomagolás *-jával!

Függvények: változó számú argumentum

Két “csillagos” (*variadic*) paraméternek nem lenne értelme (melyik kapná melyik argumentumot?), de pozicionális argumentumok megelőzhetnek egy ilyet. `print()` jó példa olyan függvényre, aminek variadikus és keyword argumentuma is van: az összes (tetszőleges számú) nem-keyword argumentumát kinyomtatja, a keyword argumentumai által meghatározott módon.

Például tegyük fel, hogy néha mértani ($\sqrt[n]{a_1 a_2 \cdot \dots \cdot a_n}$) átlagot akarunk számolni számtani ($\frac{a_1+a_2+\dots+a_n}{n}$) helyett. Akkor egy első, pozicionális argumentum lehetne a típus, a többi pedig maguk a számok.

Ebben a példában használni fogjuk a `functools` modulbeli `reduce` függvényt (számlista szorzatának kiszámolására), ami így működik: ha `f` kétargumentumú függvény, akkor pl.

```
reduce(f,[a1,a2,a3,a4])
```

ezt adja vissza:

```
f(f(f(a1,a2),a3),a4)
```

és ha egy harmadik, opcionális paramétert is megadunk, akkor az kerül a lista elejére, és azt adja vissza, ha a lista üres:

```
reduce(f,[a1,a2,a3,a4],a)
```

ezt adja vissza

```
f(f(f(f(a,a1),a2),a3),a4)
```

Függvények: változó számú argumentum

```
>>> from functools import reduce

>>> def prod(lst):
...     return reduce(lambda x, y: x*y,lst,1)
...
>>> #vagy
>>> def prod(lst):
...     res = 1
...     for i in lst: res*=i
...     return res
...
```

Függvények: változó számú argumentum

```
>>> def avg(typ, *nums):
...     return \
...         prod(nums)**(1/len(nums)) if typ == 'g' \
...            else sum(nums)/len(nums)
...
>>> avg('naposcsibe',2,3,5,9,6)
5.0
>>> avg('g',2,3,5,9,6)
4.384327654865777
```

Az első argumentum a `typ` paraméterbe került, a többi a `nums`-ba.

Függvények: változó számú argumentum

Ez a függvény jó illusztráció arra, hogy mi történik az argumentumokkal ha a variadikusak előtt pozicionálisak is vannak, de van mit szépíteni rajta. Például, mivel általában számtani közepet akarunk számolni, természetes lenne ezt alapértelmezésként megadni. Csakhogy alapértelmezett argumentum után nem jöhet variadikus, úgyhogy megcseréljük a sorrendet:

```
>>> def avg(*nums, typ='a'):
...     return sum(nums)/len(nums) if typ == 'a' \
...         else prod(nums)**(1/len(nums))
...
>>> avg(2,3,5,9,6)
5.0
>>> avg(2,3,5,9,6,typ='g')
4.384327654865777
```

Függvények: változó számú argumentum

Ebben az esetben a Python tudja, hogy hol kell abbahagynia az argumentumok gyűjtését a `nums`-ba, mert felismeri a keyword argumentumot az egyenlőségjelből. És ez egy "kötelező" keyword argumentum, mert egy variadikus után áll. (A fenti $*$, amelyet arra használtunk, hogy a következő paramétereket keyword paraméterekké kényszerítsük, úgy is felfogható, mint ennek egy speciális esete: egy "üres variadikus paraméter", amely pontosan nulla argumentumot fogad el, és amelynek egyetlen értelme az, hogy a következő paraméterek csak keyword argumentumok lehessenek.)

Függvények: változó számú argumentum

A variadikus argumentumoknak van egy "inverze" is: ha `lst` `lista` vagy `tuple`, akkor `f(*lst)` `lst` tagjaival hívja `f`-et. Például

```
>>> (lambda x,y: x+y)(*[1,2])
3
```

és

```
>>> avg(2,3,5,9,6)
5.0
```

helyett hívhatjuk így is az `avg()` függvényünket:

```
>>> avg(*[2,3,5,9,6])
5.0
```

Függvények: változó számú argumentum

Egy függvény, ami a variadikus argumentumokat és ezt az "inverz"-et is használja: A `mymap()` függvény[12] első argumentuma egy *n*-argumentumú függvény, a többi pedig *n* db. lista. Az eredmény a függvénynek a listák egymást követő elemeire való alkalmazásának eredményeiből álló lista.

Például:

```
>>> mymap(lambda x,y,z: (y,z,x), [1,2,3],[4,5,6],['a','b','c'])
[(4, 'a', 1), (5, 'b', 2), (6, 'c', 3)]
```

[12] a beépített `map()` függvény egyszerűsített változata

Függvények: változó számú argumentum

```
def mymap(fn, *lists):
    return [fn(*i) for i in  zip(*lists)]
```

Szükségünk van `*`-ra a `lists` paraméter előtt, mert nem tudjuk előre `fn` aritását, és így azt sem, hogy hány listával hívják majd `mymap()`-ot. A függvény testében `lists` értéke egy listákból álló `tuple`. Ezeket a listákat különálló argumentumként (ez `*` szerepe `zip(*lists)`-ben) adjuk meg a `zip()` függvénynek (ami szerencsére szintén tetszőleges számú argumentumot fogad el), ami `tuple`-k egy listáját[13] ad vissza: az első (ami az első iterációban az `i` értéke lesz) a `listá`k első tagjaiból áll, a második a második tagjaiból, és így tovább. És `fn` minden egyes iterációban az `i` tagjaival lesz meghívva, tehát az első iterációban a listák első tagjaival, a másodikban a listák második tagjaival, stb. Az eredmény az `fn` által visszaadott eredmények listája.

[13] valójában egy *iterable*-t, de ez most mindegy

Függvények: anoním függvények

Ezekről már mindent tudunk, ez itt csak egy kis propaganda:

- `lambda`-k, mint szinte minden, nem nélkülözhetetlenek:

  `lambda v1,...,vn: expr`

  helyett mindig írhatnánk `f`-et, feltéve, hogy azt előbb így definiáltuk:

```
def f(v1,...,vn):
    return expr
```

- Legalábbis ha a programunk amúgy nem használ `f` nevű függvényt (amit nehéz garantálni).
- Egy `lambda` definíciója ott jelenik meg, ahol használjuk.
- Ha normál függvénnyel helyettesítjük, annak definíciója esetleg nagyon máshol, mert a függvénydefiníció egy parancs, nem kifejezés, tehát nem állhat pl. egy listában.

Függvények: magasabbrendű függvények

Azokat a függvényeket nevezzük magasabbrendűeknek, amelyeket függvény argumentumokkal lehet hívni (mint pl. `sorted()`) és/vagy függvényt adnak vissza.

A magasabb rendű függvények egyik felhasználási módja a kódduplikáció elkerülése. Tegyük fel például, hogy különböző műveleteket kell végeznünk számlistákon. Mindegyik művelethez írhatunk függvényeket:

```
>>> def inc_list(l):
...     return [x+1 for x in l]
...
>>> def double_list(l):
...     return [2*x for x in l]
...
>>> inc_list(list(range(10)))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> double_list(list(range(10)))
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Függvények: magasabbrendű függvények

De ez a két függvény gyakorlatilag ugyanaz; az egyetlen különbség az, hogy mit csinálnak a lista tagjaival. Így van értelme, hogy *azt* extra paraméterré alakítsuk:

```
>>> def process_list(fun,l):
...     return [fun(x) for x in l]
...
>>> process_list(lambda x: x+1, list(range(10)))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> process_list(lambda x: 2*x, list(range(10)))
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

`process_list()` valójában a korábban (itt) látott `mymap()` függvény egyszerűsített változata (ami meg a beépített `map()` függvényé).

A beépített `map()` függvény egy iterálhatót ad vissza, nem `list`át, de ez `list`ává alakítható, ha arra volna szükségünk:

```
>>> list(map(lambda x: x**2,range(10)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Függvények: magasabbrendű függvények

Íme egy példa függvényt visszaadó függvényre:

```
>>> def compose(f1,f2):
...     return lambda x: (f1(f2(x)))
...
>>> compose(lambda x: x+1, lambda x: 2*x)(4)
9

>>> compose(lambda x: 2*x, lambda x: x+1)(4)
10

>>> from math import sqrt

>>> compose(sqrt, lambda x: x+1)(1)
1.4142135623730951
```

Függvények: dokumentáció

Egy függvény teste parancsok és kifejezések sorozata, amik megjelenésük sorrendjében hajtódnak végre/értékelődnek ki. Következésképp nem számít, ha ezek közé beteszünk mondjuk egy `str`inget. De ha ez az első dolog a függvény testében, akkor Python *documentation string*-nek (*docstring*) tekinti, és eltárolja a függvény `__doc__` *attribútum*ában.

```
>>> def fun():
...     """
...     Ez a függvény kiválóan tesz semmit.
...     Használata: fun()
...     """
...     pass
...
```

Függvények: dokumentáció

```
>>> print(fun.__doc__)

    Ez a függvény kiválóan tesz semmit.
    Használata: fun()



>>> help(fun)
Help on function fun:

fun()
    Ez a függvény kiválóan tesz semmit.
    Használata: fun()
```

Ezt aztán kérésre megmutatják a különféle fejlesztői környezetek. Pl. IPython akkor ha `fun?`-t vagy `help(fun)`-t írunk.

Dokumentáljuk íly módon a kicsit is bonyolultabb dolgot művelő függvényeinket!

Modulok

Modulok

A Python néhány kevésbé használt része és a mások (akik mi is lehetünk) által írt függvények (és egyebek, pl. osztályok) alapértelmezés szerint nem töltődnek be. Ezeket *modulok*ban gyűjtik össze, amelyeket az alábbi módokon lehet a programunk számára elérhetővé tenni (`importálni`).

1. `import math` Ez minden `math` modulbelidolgot importál; ezeket a nevük elé írt `math.`-al érhetjük el. Például `math.sqrt()`.
2. `import math as mt` Mint az előző, csak `mt`-t (egy *alias*-t) írhatunk `math` helyett. Tehát pl. `mt.sqrt()`-t.
3. `from math import sqrt` Ez csak az `sqrt()` függvényt importálja a `math` modulból, de azt elérhetővé teszi a `math.` prefix nélkül. Egyszerre több függvényt (és egyebeket) is importálhatunk, így:

   ```
   from math import sqrt, isqrt
   ```

Modulok

Egy modul dokumentációjához hozzáférhetünk így:

```
>>> help("math")
```

vagy így: `math?` IPython-ban. Így kapunk egy összefoglalót a modulról, valamint egy felsorolást a benne elérhető függvényekről (osztályokról, egyebekről). De ehhez előbb importálni kell a megfelelő modult.

Objektumorientált programozás

Tegyük fel, hogy szükségünk van egy adattípusra mátrixokkal való számoláshoz. A mátrixokat könnyen ábrázolhatjuk egymásba ágyazott listákkal: például a sorok listájával, ahol egy sor a tagjainak listájával van reprezentálva. Tehát a `[[1,0,0],[0,1,0],[0,0,1]]` reprezentálná a 3 3-as egységmátrixot. Ha `m` ilyen mátrix, akkor i. sora j. tagjához így férhetünk hozzá vagy módosíthatjuk: `m[i][j]` vagy `m[i][j] = v`.

Ez működik, de az absztrakció hiánya (listákkal kell foglalkoznunk mátrixok helyett) mindenféle nehézségekhez vezet: pl. nem változtathatjuk meg a reprezentációt, anélkül, hohu minden kódot, ami mátrixokkal foglalkozik, meg kellene változtatnunk. Márpedig később rájöhetünk, hogy ritka mátrixokkal kell foglalkoznunk (olyanokkal, ahol majdnem minden elem ugyanaz): ezeknek a fenti módon való reprezentálása roppant nagy helyveszteség.

Ennek (és néhány más) problémának egy lehetséges megoldása egy mátrix *osztály* definiálása.

```
class Mtx():

    def __init__(self, list):
        nc = len(list[0])
        #a sorok azonos hosszúságúak
        assert all([len(l) == nc for l in list[1:]]),\
        'eltérő hosszúságú sorok'
        self._list = list
        self._no_of_rows = len(list)
        self._no_of_columns = nc

    def no_of_rows(self): return self._no_of_rows

    def no_of_cols(self): return self._no_of_columns

    def get_row(self,rn): return self._list[rn]
```

```
def get_col(self,cn): return [l[cn] for l in self._list]

def get(self,r,c): return self._list[r][c]

def set(self,r,c,value): self._list[r][c] = value

def __repr__(self):
    return f'{self._no_of_rows} times {self._no_of_columns}
    Mtx: {self._list}'

def __str__(self):
    s = ""
    for i in self._list:
        s += str(i)+'\n'
    return s
```

1. Az osztály definíciója hasonlít egy függvény definíciójához: egy blokk, aminek az első sorát egy kulcsszóval kezdjük (ami ebben az esetben `class`), azután az osztály neve, majd kettőspont következik. (Az osztálynév és a kettőspont között állhat zárójelek között más osztályok neveinek vesszővel elválasztott listája. Később látni fogunk erre példát.)
2. A definíció testében a `def`-ek pontosan úgy néznek ki, mint a függvénydefiníciók, de ezek metódusokat definiálnak, nem függvényeket.
3. Egy metódus első argumentumának (amelyet szokás `self`-nek hívni) az osztály azon példánya lesz az értéke, amelyre a metódust meghívjuk. Vagyis ha `m` az `Mtx` egy példánya, akkor `m.get_row()` a `.get_row()` metódust úgy hívja meg, hogy a testében `self` értéke `m` lesz. (A `.` szerepe az `m.get_row()` hívásban ugyanaz, mint pl. `l.append(42)`-ben.)

4. A “mágikus” `.__init__()` metódus akkor fut, amikor a `Mtx()` meghívásával létrehozzuk `Mtx` egy példányát (mostantól: egy `Mtx`-et). Ez a mágia: nem nekünk kell hívni. Az újonnan létrehozott példány lesz az `.__init__()` első argumentuma, és a `Mtx` argumentumai (ez esetben csak egy) a többi.
   Pl. a következőben

   ```
   >>> m = Mtx([[1,2],[3,4]])
   ```

   `.__init__()` második argumentuma `[[1,2],[3,4]]`, az első pedig az új példány, amely ebben az esetben a `m` változóhoz is hozzá lesz rendelve.
   `m` értéke valóban egy `Mtx`:

   ```
   >>> isinstance(m, Mtx)
   True
   ```

5. Az `.__init__()`-beli `assert` utasítás megakadályoz egy tipikus hibát:

```
>>> m = Mtx([[1,2,3],[4,5,6],[7,8]]) #nem fog menni
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in __init__
AssertionError: eltérő hosszúságú sorok
```

6. `_list`, `_no_of_rows` és `_no_of_columns` az osztály *attribútumai* (*mezői*, *tulajdonságai*). Ezek példányspecifikus adatokat tartalmaznak (szemben a metódusokkal, amelyek az osztályhoz tartoznak). Az aláhúzás azt jelzi, hogy bár lehet, nem illik közvetlenül hozzájuk férni (olvasni vagy beállítani) őket, csak az osztály (vagy annak *alosztályai* (*gyermekei*), ld. később!) metódusaival. Tehát ez:

```
>>> m._no_of_columns
2
```

példa arra, amit nem szabadna tennünk az osztály definícióján kívül. Az oszlopok száma lekérdezésének az "accessor" ("getter") metódus meghívása:

```
>>> m.no_of_cols()
2
```

a helyes módja.

7. Két másik mágikus metódus, a `.__str__()` és a `.__repr__()` határozza meg, hogy az osztály példányai hogyan lesznek kinyomtatva és hogyan jelennek meg pl. a debuggerben. Itt ezek hívódnak implicit módon:

```
>>> m # ld. .__repr__() definícióját!
2 times 2 Mtx: [[1, 2], [3, 4]]

>>> print(m) # ld. .__str__() definícióját!
[1, 2]
[3, 4]



>>> m.set(1,1,-5); print(m)
[1, 2]
[3, -5]
```

8. Mindvégig az `assert`-et használjuk annak ellenőrzésére, hogy bizonyos feltételek teljesülnek. Ez, mint már korábban volt róla szó, nem feltétlenül jó ötlet; de mivel nem tudjuk, hogyan definiáljunk kivételeket Pythonban (valójában épp most tanuljuk meg, mivel a kivételek a az `Exception` osztályból származtatott osztályok), így most ez a legjobb, amit tehetünk.

Ahhoz, hogy az `Mtx` osztály hasznos legyen, legalább mátrix-szorzásra és -összeadásra meg kell tanítanunk. Ezért definiáljuk benne az `.add()` és `.prod()` metódusokat, például így:

```
def add(self, other):
    assert isinstance(other,Mtx),\
       'csak Mtx-et lehet Mtx-hez adni'
    assert (self.no_of_cols() == other.no_of_cols()
       and self.no_of_rows() == other.no_of_rows()),\
            'eltérő hosszúságú sorok'
    nr = self.no_of_rows()
    nc = self.no_of_cols()
    m = Mtx([nc * [0] for _ in range(nr)])
    for i in range(nr):
        for j in range(nc):
            m.set(i,j,self.get(i,j)+other.get(i,j))
    return m
```

```
def prod(self, other):
    assert isinstance(other,Mtx),\
     'Mtx-et csak Mtx-el lehet szorozni'
    assert self.no_of_cols() == other.no_of_rows(),\
     f'''{self} oszlopai ({self.no_of_cols()}) és
     {other} sorainak száma ({self.no_of_rows()}) különbözik.'''
    nr = self.no_of_rows()
    nc = other.no_of_cols()
    onr = other.no_of_rows()
    m = Mtx([nc * [0] for _ in range(nr)])
    for i in range(nr):
        for j in range(nc):
            m.set(i,j,sum(
              [self.get(i,k)*other.get(k,j)\
                for k in range(onr)]))
    return m
```

Ezek az osztálydefiníció részei kell legyenek, különben a Python nem tudná, hogy melyik osztály metódusai.

```
>>> m1 = Mtx([[1,2,3]]); m2 = Mtx([[1,2],[3,3],[2,1]])
>>> m1.add(42) #hibát kell dobjon
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 37, in add
AssertionError: csak Mtx-et lehet Mtx-hez adni
>>> print(m1.add(m1))
[2, 4, 6]
```

```
>>> print(m2.prod(m1)) #hibát kell dobjon
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 49, in prod
AssertionError: [1, 2]
[3, 3]
[2, 1]
 oszlopai (2) és
[1, 2, 3]
 sorainak száma (3) különbözik.
>>> print(m1.prod(m2))
[13, 11]
```

Mit tehetünk még `Mtx`-ekkel az összeadáson és szorzáson kívül? Definiálhatnánk például egy olyan metódust, amely skalárral való szorzást végez. (Definiáljunk!) Determináns is hasznos lenne, de annak csak négyzetes mátrixok esetén van értelme. Ugyanez vonatkozik a hatványokra is.

Természetesen csinálhatnánk `.power()` vagy `.determinant()` metódusokat `Mtx`-ben, és kezdhetnénk a definíciójukat `_no_of_columns == _no_of_rows` ellenőrzésével; de sokkal jobb, ha létrehozzuk `Mtx` egy speciális változatát, mondjuk `SqMtx`-t, és erre definiálunk `.power()` és `.determinant()` metódusokat. `SqMtx`-et `Mtx` *alosztályának* vagy *gyermekének* nevezzük, `Mtx` pedig az `SqMtx` *szülője*.

Egy alosztály, mint például a `SqMtx`, *örököl* mindent (attribútumok, metódusok) a szuperosztálytól, kivéve, amit *felülír* a definíciójában; a `SqMtx` esetében ezek a `.__init__()` és a `__repr__()` metódusok lesznek.

Az `SqMtx` egy lehetséges megvalósítása (a `.determinant()` megírását meghagyom (nem triviális) gyakorlatnak):

```
class SqMtx(Mtx):
    def __init__(self, list):
        super().__init__(list)
        assert self._no_of_columns == self._no_of_rows, \
          f'A sorok ({self._no_of_rows}) és oszlopok száma \
          ({self._no_of_columns}) különbözik.'
        self._dim = self._no_of_rows
    def power(self,n):
        assert isinstance(n,int), 'A kitevő nem egész'
        res = self
        while n>1:
            res = res.prod(self)
            n -= 1
        return res
    def __repr__(self):
        return f'SqMtx of dimension {self._dim}: {self._list}'
```

```
>>> sm = SqMtx([[1,2],[3,4]])
>>> isinstance(sm, Mtx) #minden SqMtx Mtx is
True
>>> sm.add(sm) #ezért ^ működik ez is
2 times 2 Mtx: [[2, 4], [6, 8]]
>>> print(sm.power(1)); print(sm.power(3))
[1, 2]
[3, 4]

[37, 54]
[81, 118]
```

A definíció első sora deklarálja a szülő(ke)t. (Ha több is van, vesszővel választjuk el őket.) A `__repr__()` metódus definíciója felülírja a szülőben megadott definíciót. Az `.__init__()` metódussal a helyzet részben hasonló, amennyiben a definíció felülírja az `Mtx .__init__()` metódusát. A különbség az, hogy használja is azt. Ebben a kulcs a `super()` hívása, amely visszaad egy hivatkozást az objektum `Mtx` részéhez; tehát

```
super().__init__(list)
```

inicializál egy `Mtx`-t. Ha ez megtörtént, akkor elvégezzük a többit, azaz az inicializálás `SqMtx`-specifikus részét.