Algoritmizálás és programozási nyelv használata

Eddig jutottunk eddig

Könyvünk előző kötetében belekóstoltunk a programozásba, és elég sok mindent megtanultunk – először ezeket ismételjük át.

Mindenhol programok

Tudjuk már, hogy a háztartási gépektől kezdve az autókon és a repülőgépeken keresztül a robotokig mindenütt van számítógép, és ahol számítógépek, ott programok is vannak. A digitáliskultúra-órákon a bennünket jobban érdeklő, hagyományos értelemben vett számítógépek (laptopok, asztali gépek, szerverek) és mobileszközök a bekapcsolásukkor egy fő programot indítanak el, az operációs rendszert.

A többi program elindítása, futásuk közben az eszköz erőforrásaihoz (perifériák, háttértárak, memória, processzor) való hozzáférés szabályozása és a programok megállítása az operációs rendszer feladata. A programok egy része automatikusan indul, más részüket a felhasználó indítja el.

Kérdések

Milyen operációs rendszer fut a számítógépeden és milyen a mobileszközeiden?
Milyen háttértár van a számítógépedben, milyen a mobileszközeidben?

A programok elindításukig csak a háttértáron találhatók meg. Elindításkor a memóriába tölti őket a számítógép, és a processzor megkezdi a végrehajtásukat. A programot tartalmazó fájl természetesen megmarad a háttértáron ilyenkor is. A programok elindítása történhet az ikonjukra való kattintással, az ikon ujjunkkal történő megérintésével, de minden program elindítható a számítógép parancssorából is.

Nyissuk meg a számítógépünk parancssorát, és indítsunk el innen egy szövegszerkesztőt, egy képszerkesztőt, egy böngészőprogramot!

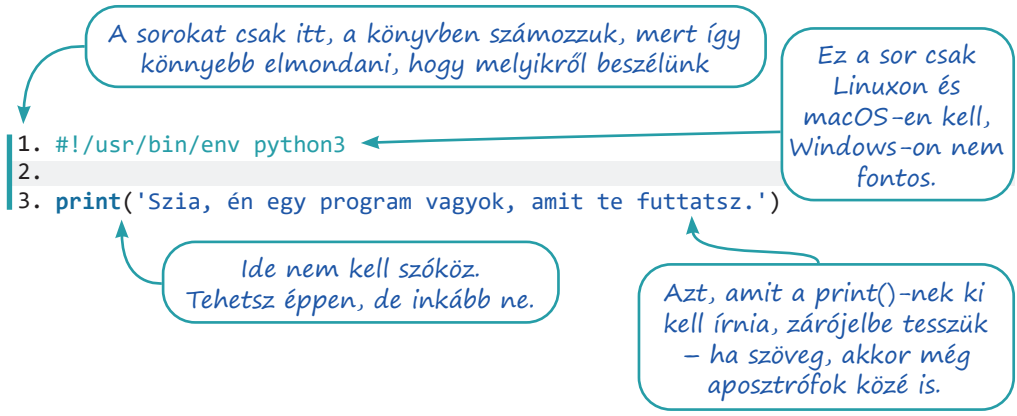
Forráskód, programozási nyelvek és fejlesztői környezet

Programjainkat az esetek túlnyomó többségében forráskódként fogalmazzuk meg. A forráskód az angol nyelvből vett szavakon kívül rendszerint szép számban tartalmaz még mindenféle egyéb jeleket és számokat, és hellyel-közzel mindenki el tudja ezeket olvasni – a programozáshoz értők nyilván lényegesen eredményesebben.

A forráskódot sima szöveges fájlokban tároljuk, és a fájl kiterjesztése általában a programozási nyelvre utal. Egy köszönést a képernyőre író egyszerű program neve Ruby nyelv használata esetén lehet `szia.rb`, C++ nyelven dolgozva a fájlnév alighanem a `szia.cpp` formát ölti, és ha – mint ahogy az előző kötetben és ebben is – Pythont használunk, akkor a fájl neve esélyesen `szia.py`.

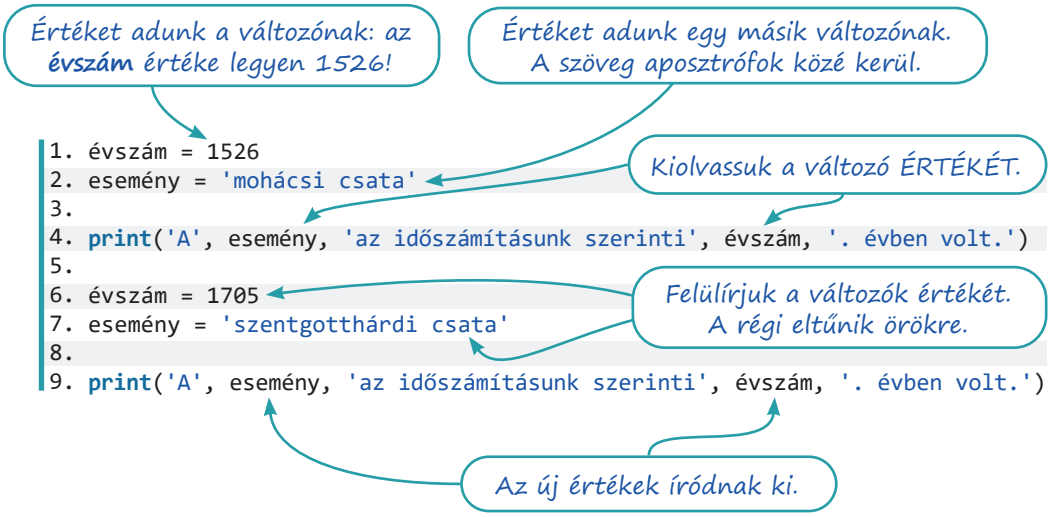
A programjainkat a legegyszerűbb szerkesztőprogramban is írhatjuk, de általában valamilyen fejlesztői környezetet, azaz IDE-t használunk. A legegyszerűbb IDE-ket „csak” az különbözteti meg az egyszerű szerkesztőktől, hogy színezéssel segítik a programozót a programban való jobb eligazodásban. Vannak ennél lényegesen nagyobb tudású és ennek megfelelően bonyolultan használható IDE-k is.

Nyissunk meg egy IDE-t, és írjuk meg benne azt a programot, amelyik elárulja, hogy épp egy programot futtatunk:



Változók

A programjainknak gyakran kell adatokat tárolniuk. Az adatokat a gép memóriájába teszik el, és hogy a memórián belül pontosan hova, azt a legtöbbször nem tudjuk. Az eltett adatokat úgy tudjuk ismét elővenni, ha megadjuk azt a nevet, amit az eltett adathoz hozzárendelünk. Ezt a hozzárendelt nevet változónak hívjuk, és az eltárolást programozóul úgy mondjuk, hogy értéket adunk a változónak. Az értékadás egy művelet, ugyanúgy, mint az összeadás vagy az osztás, és van műveleti jele is, ami sok programozási nyelvben – a Pythonban is – az egyenlőségjel.



A fenti kód kimenete csúnyácska, hiszen amikor a `print()` a vesszővel elválasztott kiírandó elemeket összefűzi, szóközt is tesz, és így a pont nem kerül pontosan az évszám mellé. A megoldást is ismerjük már, hasonlítsuk össze az alábbi két sort:

```
1. print('A', esemény, 'az időszámításunk szerinti', évszám, '. évben volt.')
2. print('A ', esemény, ' az időszámításunk szerinti ', évszám, '. évben volt.', sep='')
```

Ez itt két aposztróf, köztük pedig nincs semmi.

A második sor végén, a `sep=''` részben, a két aposztróf közé kerül az a karaktersorozat, amit a `print()` a vesszővel összefűzött részek közé ír. Alapértelmezetten ez egy szóköz, de mi most semmit sem írunk közéjük, azaz nincs elválasztó karakter, és a pont végre az évszám mellé kerül. Viszont így sehol máshol sem lesz szóköz a sorban, nekünk kell beírni mindenütt.

Változók típusai, típusátalakítás, adatbekérés

A felhasználótól adatot az `input()` utasítással kérhetünk be. A bekért eredményt változóban tárolhatjuk. Az `input()` nem olyan ügyes, mint a `print()`:, nem tud összefűzni több kiírandó részt. Szerencsére az összeadást a Python karaktersorozatokra is tudja értelmezni: az `'el' + 'iramlik'` művelet eredménye `'eliramlik'`. Az összeadással azonban baj lehet, ha egy számot és egy szöveget adunk össze. Mennyi a `2 + 'asztal'`?

Az `str()` (string, azaz karaktersorozat) utasítással a zárójelben lévő számot szöveggé alakíthatjuk, és `'2' + 'asztal'` már a Python számára is `'2asztal'`.

Ez is értékadás: amit az input() a felhasználótól kap, azt betesszük a szorzó nevű változóba.

```
1. szorzandó = 5
2. szorzó = input('Mennyivel szorozzam meg az ' + str(szorzandó) + '-öt? ')
3. print(szorzandó, '-ször ', szorzó, ' annyi mint ', sep='', end='')
4. szorzó = int(szorzó)
5. print(szorzandó * szorzó)
```

Itt azt mondjuk meg a print()-nek, hogy még ne kezdjen új sort.

Amit az input() a felhasználótól kap, azt mindig szövegként adja át, még a számokat is. Ha matekozni akarunk vele, akkor például egész számmá (integer, azaz int) kell alakítanunk.

Eddig összesen négyféle típusú elemi adatot tároltunk változóban:

- karaktersorozatot, más néven szöveget;
- egész számot;
- logikai értéket (az ilyen változók értéke `True` [igaz] vagy `False` [hamis] lehet);
- és ritkábban tizedestörtet, más néven lebegőpontos számot (persze tizedesponttal a vessző helyett).

Elágazások

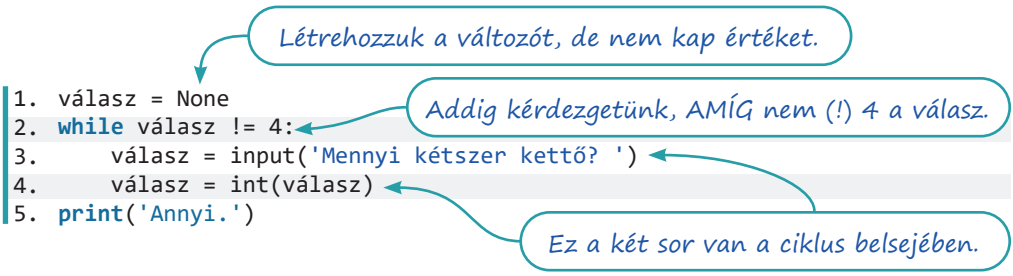
Nagyon hamar felmerül az igény, hogy a programunk eltérő feltételek esetén másként viselkedjen. Például, ha elmúlt este nyolc, váltson sötét témára a telefon, ha helyesen adta meg a jelszót a felhasználó, akkor engedjük belépni. Az ilyen problémák megoldására való az `if` és az `else` utasítás.

Mit csinál az alábbi program?


```
1. ellenség = input('Ki volt a Piroska nevű szuperhős főellensége? ')
2. if ellenség == 'farkas' or ellenség == 'Farkas':
3.     print('Okos vagy.')
4.     print('Nem is kicsit.')
5. else:
6.     print('Hááát....')
7.     print('Nem.')
8. print('Legközelebb a hét törpét kérdezem - visszafelé!')
```

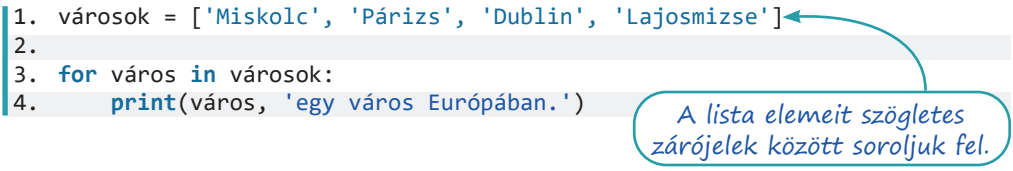
Ciklusok és listák

Ha egy feladatrész ismétlődik, akkor ciklus használatával oldjuk meg. Van feltételes és bejárós ciklusunk. A feltételes ciklus magja addig ismétlődik, amíg fennáll a ciklus elején megfogalmazott feltétel. Az amíg (angolul `while`) a feltételes ciklus elejét jelző utasítás.




```
1. válasz = None
2. while válasz != 4:
3.     válasz = input('Mennyi kétszer kettő? ')
4.     válasz = int(válasz)
5. print('Annyi.')
```

Az összetartozó adatokat listában tároljuk, a listákat pedig bejárós ciklussal tudjuk bejárni. A bejárós ciklus ciklusváltozója (a lenti példában a város) mindig a lista aktuális elemét tartalmazza. A bejárós ciklus másik neve: `for`-ciklus.




```
1. városok = ['Miskolc', 'Párizs', 'Dublin', 'Lajosmizse']
2.
3. for város in városok:
4.     print(város, 'egy város Európában.')
```

Elemi adattípusok és elágazások

Feladatok

1. Írjuk képernyőre programmal egy általunk választott vers két versszakát! A vers előtt adjuk meg a szerzőt és a címet, majd sorkihagyást követően az első, újabb sorkihagyást követően a második versszakot írjuk ki! A programunk legfeljebb három `print()` utasítást használhat.
2. Mondatszerű leírással (más szóval: pszeudokódban) megadunk egy programot. A program bemenete egy állatfaj és az állat legnagyobb sebessége km/h-ban kifejezve.

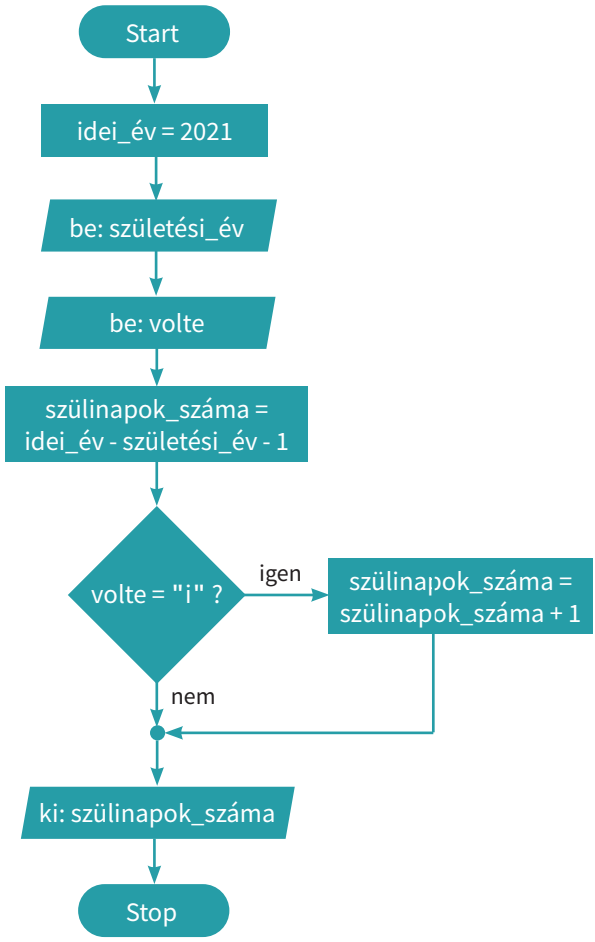
```
program
    be: állatfaj
    be: sebesség
    elágazás
        ha sebesség legfeljebb 50:
            hol := „városban”
        különbenha sebesség legfeljebb 90:
            hol := „országúton”
        különben:
            hol := „autópályán”
    elágazás vége
    ki: „Az”, állatfaj, „a legnagyobb sebességével”, hol,
        „haladhat.”
program vége
```

a. Mit csinál a program?
b. Írjuk át a mondatszerű leírást folyamatábrává!
c. Kódoljuk a programot!

```
állatfaj = input('Add meg egy állatfaj nevét! ')
sebesség = input('Add meg az állatfaj sebességét! ')
sebesség = int(sebesség)
if sebesség <= 50:
    hol = 'városban'
elif sebesség <= 90:
    hol = 'országúton'
else:
    hol = 'autópályán'
print('Az', állatfaj, 'legnagyobb sebességével', hol, 'haladhat.')
```

d. Teszteljük a program működését az interneten fellelhető, a témába vágó adatok használatával!
e. Nagyon hamar találunk olyan állatot, amely esetében hibás ítéletet hoz a programunk. Mit kell tudnia az ilyen állatnak? Hogyan javítható a programunk?

3. Kérdezzük meg a felhasználótól, hogy melyik évben született, és hogy volt-e már idén születésnapja! A két adat ismeretében írjuk ki, hogy hányadik születésnapját ünnepelte! Először készítsük el a megoldás folyamatábráját vagy mondatszerű leírását. Ha elkészültünk, írjuk meg a programkódot is.



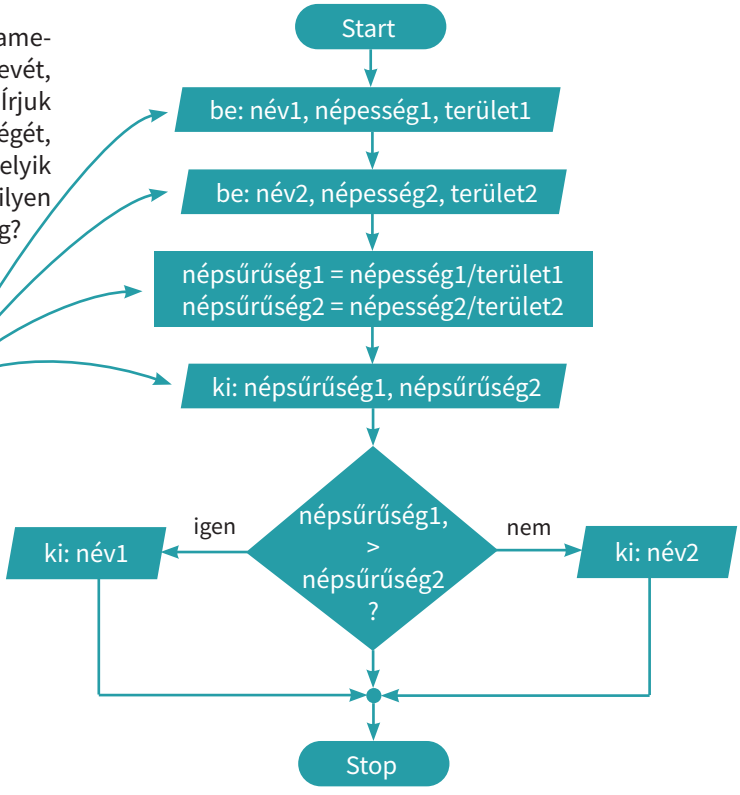

```
program
    idei_év = 2021
    be: születési_év
    be: volte
    szülinapok_száma = idei_év – születési_év - 1
    ha volte = 'i':
        szülinapok_száma = szülinapok_száma + 1
    ki: szülinapok_száma
program vége
```

```
1. idei_év = 2021
2. születési_év = input('Melyik évben születtél? ')
3. születési_év = int(születési_év)
4. volte = input('Volt már idén születésnapod? (i/n) ')
5.
6. szülinapok_száma = idei_év - születési_év - 1
7. if volte == 'i':
8.     szülinapok_száma = szülinapok_száma + 1
9. print('Utoljára a ', szülinapok_száma, '. születésnapodat ünnepelted.', sep='')
```

4. Kérdezzük meg a felhasználótól a nevét és azt, hogy a nap hányadik órájában járunk! Köszönjünk neki a nevét megemlítve a napszaknak megfelelően, reggel nyolcig „Jó reggelt”, este hatig „Jó napot”, aztán „Jó estét” kívánva!
   a. Készítsük el a program mondatszerű leírását vagy folyamatábráját!
   b. A leírás vagy az ábra alapján kódoljuk a programot!
   c. Kihívást jelentő feladat: Ne a felhasználótól kérdezzük meg az órát, hanem olvassuk ki a számítógép órájából! Az internet segíteni fog az óra kiolvasásának kódolásában.
5. Írjunk olyan programot, amelyik bekéri két autómárka nevét és az autók maximális sebességét! Írjuk ki, hogy melyik autó a gyorsabb.

```
program
    be: egyik_neve
    be: egyik_sebessége
    be: másik_neve
    be: másik_sebessége
    elágazás – ha egyik_sebessége > másik_sebessége:
        ki: egyik_neve
    különbenha másik_sebessége > egyik_sebessége:
        ki: másik_neve
    különben:
        ki: 'Egyformán gyorsak.'
    elágazás vége
program vége
```

6. Írjunk olyan programot, amelyik bekéri két ország nevét, népességét és területét! Írjuk ki a két ország népsűrűségét, és azt, hogy ez az adat melyik országban a nagyobb! Milyen típusú adat a népsűrűség?



Nem baj, ha a folyamatábrán az utasításokat alkalmasan csoportosítjuk. Az a lényeg, hogy értsük az ábrát!

7. Kérjünk be két egész számot a felhasználótól, és írjuk ki, hogy a kisebb osztója-e a nagyobbnak (azaz egész szám-e a hányados)!
   a. Készítsük el a feladatot megoldó algoritmus mondatszerű leírását!
   b. Kódoljuk az algoritmust: készítsük el a programot!
   c. A példamegoldás csak pozitív egészekkel boldogul. Módosítsuk úgy akár a példát, akár saját működő programunkat, hogy negatív számokat is megadhassunk!
   d. Módosítsuk úgy a programunkat, hogy csak a valódi osztókat minősítse osztónak!
   e. Írjuk meg a megoldást úgy, hogy az alapműveletek közül csak az összeadást és kivonást használhatjuk.

A bekért számot azonnal egésszé alakítjuk. Ha tetszik a megoldás, használd bátran!

```
1. egyik = int(input('Mi legyen az egyik szám? '))
2. másik = int(input('Mi legyen a másik szám? '))
3.
4. # mindig a nagyobb szám legyen az osztandó
5. if egyik >= másik:
6.     osztandó = egyik
7.     osztó = másik
8. else:
9.     osztandó = másik
10.    osztó = egyik
11.
12. hányados = osztandó / osztó
13. if hányados == int(hányados):
14.     print(osztó, ' osztója ', osztandó, '-nek.', sep='')
15. else:
16.     print(osztó, ' nem osztója ', osztandó, '-nek.', sep='')
```

A kettőskereszttel kezdődő sort a Python figyelmen kívül hagyja. Így írunk megjegyzéseket a kódba.

Ha a szám egésszé alakítva ugyanaz marad, akkor már eleve egész volt.

Mint a legtöbb programozási nyelvben, a Pythonban is van olyan osztás, amelyik a maradékot adja meg. Ez az úgynevezett moduloosztás, vagy egyszerűbben csak mod, és Pythonban a jele a %. Így írjuk le a műveletet: 9 % 4 = 1, azaz kilencet néggyel osztva egy a maradék. A fenti program 12–13. sora helyett tehát használható az

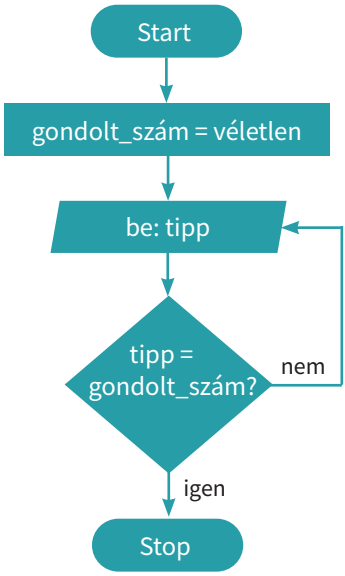
```
if osztandó % osztó == 0:
```

sor is.

Ciklusok és listák

Feladatok

1. Gondoljon a programunk egy számra egy és tíz között! A felhasználó feladata a szám kitalálása. Addig próbálkozhat, amíg el nem találja. A program folyamatábrával és mondatszerű leírással:




```
program
    gondolt_szám := véletlen(10)
    tipp := speciális érték
    ciklus amíg tipp <> gondolt_szám:
        be: tipp
    ciklus vége
program vége
```

Ide valami olyan értéket írunk, ami biztosan NEM a jó megoldás. Lehet pl. –1, de Pythonban a None a bevett szokás.

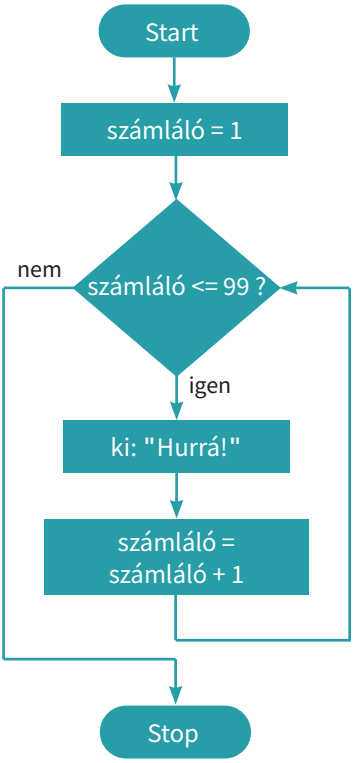
a. Hogyan állítunk elő véletlen számot? Keressük meg az interneten (angolul több találatunk lesz, keressünk a *random integer Python* kifejezésre)!
b. Kódoljuk az algoritmust!

```
1. import random
2.
3. gondolt_szám = random.randint(1,10)
4. tipp = None
5. while tipp != gondolt_szám:
6.     tipp = input('Tippelj! ')
7.     tipp = int(tipp)
```

c. Javítsunk annyit a programon, hogy találat esetén dicsérje meg a felhasználót!
d. Módosítsuk úgy a programot, hogy írja ki a felhasználónak, hogy a hibás tipp kisebb vagy nagyobb a gondolt számnál!

2. Vegyünk fel a programunkba egy listát: `['barack', 'körte', 'dinnye', 'narancs']`! Írjuk ki bejárós ciklussal mindegyikről, hogy egy gyümölcs!

3. Írjuk ki kilencvenkilencszer, hogy „Hurrá!”
   a. A kiírást először feltételes, azaz `while`-ciklussal végezzük el. Szükségünk lesz egy számláló nevű változóra, aminek a ciklusba való belépés előtt az 1 értéket adjuk, majd minden kiírást követően növeljük az értékét a cikluson belül. A ciklusba való belépés feltétele az, hogy a számláló értéke ne legyen nagyobb 99-nél.




```
számláló = 1
ciklus amíg számláló <= 99:
    ki: „Hurrá!”
    számláló := számláló + 1
ciklus vége
```

```
1. számláló = 1
2. while számláló <= 99:
3.     print('Hurrá!')
4.     számláló = számláló + 1
```

   b. Megoldjuk a feladatot bejárós, azaz `for`-ciklussal is. A `for`-nak be kell járnia valamit, ami 99 elemű – ilyen bejárható objektumot készíthetünk a `range(99)` utasítással. Az utasítás használatát keressük meg interneten!

```
ciklus 99-szer:
    ki: „Hurrá!”
ciklus vége
```

A mondatszerű leírásban élhetünk alkalmas egyszerűsítéssel – elvégre csak nekünk kell érteni!

```
1. for _ in range(99):
2.     print('Hurrá!')
```

Pythonban az alávonást szokás ciklusváltozóként használni, ha a változót a ciklusmagban nem használjuk. Ez csak szokás, nem szabály!

   c. Írjuk át mindkét programunkat úgy, hogy ne legyen sortörés a szavak után, csak miután kiírtuk mind a kilencvenkilencet!

4. Írjuk ki az egy és száz közötti hárommal osztható számokat! Elegáns megoldás volna a `for szám in range(3,101,3)` utasítás használata, de most tekintsünk el tőle – így többet tanulunk az algoritmusokról.
   a. Írjuk meg a programot mondatszerű leírással!
   b. Kódoljuk a programot! Segít, ha tudjuk, hogy létezik a maradékképző osztás, aminek a jele: „%”. A művelet eredménye az osztás maradéka, azaz
   9%3 = 0 és 9%2 = 1

c. Számoljuk meg, és írjuk ki, hogy hány hárommal osztható számot találunk!
d. Módosítsuk úgy a programot, hogy egy és száz helyett a felhasználó által megadott számokat használjuk!
e. Írjuk át úgy a programot, hogy a felhasználó mondhasson számot a három helyett!
f. Írjuk meg a programot a másik ciklus felhasználásával!

A megoldás bejárós ciklussal:

```
1. eleje = input('Honnan induljunk? ')
2. eleje = int(eleje)
3. vége = input('Meddig számoljunk el? ')
4. vége = int(vége)
5. osztó = input('Hánnyal osztható számokat keresünk? ')
6. osztó = int(osztó)
7. ennyi_ilyen_van = 0
8.
9. for szám in range(eleje,vége+1):
10.     if szám % osztó == 0:
11.         print('A', szám, 'osztható ezzel:', osztó)
12.         ennyi_ilyen_van = ennyi_ilyen_van + 1
13.
14. print(ennyi_ilyen_van, 'ilyen számot találtam.')
```

A feltételes ciklust használó megoldásban a 9-től 12-ig számozott sorokat cseréljük ki az alábbiakra:

```
9. szám = eleje
10. while szám <= vége:
11.     if szám % osztó == 0:
12.         print('A', szám, 'osztható ezzel:', osztó)
13.         ennyi_ilyen_van = ennyi_ilyen_van + 1
14.     szám = szám + 1
```

5. Állítsunk elő egy ezer és tízezer közötti egész számokat tartalmazó, húszelemű listát! A lista elemei azoknak a járműveknek a tömegét adják meg, amiket ma egy komphajó átvitt a folyón. Nehéznek számítanak a 9300 kilónál nehezebb járművek. Írjunk programot, ami válaszol a következő kérdésekre:
   a. Volt-e olyan jármű ma a hajón, ami nehéznek számít? Írjuk ki, ha volt ilyen!
   b. Hány ilyen jármű volt?
   c. Hány kiló járművet vitt át a komp ma összesen?
   d. Mennyi a ma átvitt, nehéznek számító járművek össztömege?
   e. Ha a „nehéz” holnaptól nem 9300, hanem 9000 kilogramm, hány helyen kell átírni a programot? Mit kell tennünk, ha azt szeretnénk, hogy az ilyen változások egyszerűen, egyetlen helyen való átírást jelentsenek?

```
import random

tömegek = []
for _ in range(20):
    tömegek.append(random.randint(1000,10000))
# kész a listánk
print(tömegek) #csak hogy könnyű legyen ellenőrizni

volt_nehéz = False
nehezek_száma = 0
össztömeg = 0
nehezek_össztömege = 0
for tömeg in tömegek:
    össztömeg = össztömeg + tömeg
    if tömeg > 9300:
        volt_nehéz = True
        nehezek_száma += 1
        nehezek_össztömege += tömeg

if volt_nehéz:
    print('Volt 9300 kilónál nehezebb jármű.')

print(nehezek_száma, '9300 kilónál nehezebb jármű volt.')
print('Ma', össztömeg, 'kilónyi járművet vitt át a komp.')
print('Ebből nehéznek összesen', nehezek_össztömege, 'kilónyi számít.')
```

Ha nem használjuk a ciklusváltozót, adhatunk neki semmitmondó nevet.

A változók értékét többféleképp is növelhetjük.

Jó úgy is, hogy
If volt_nehéz == True:
de így könnyebben olvasható.

Szövegek, eljárások, függvények

Mit tudunk eddig a szövegekről?

Szövegeket (karakterláncokat, stringeket) már használtunk programjainkban. Azt is tudjuk, hogy két szöveg összeadható egy összeadásjellel, és az összeadás valójában összefűzést, egymás mellé írást jelent. A csak számokat tartalmazó karakterláncokból tudunk egész vagy valós számot csinálni (az `int`() vagy a `float`() utasítással), és a számokat is át tudjuk alakítani szöveggé (az `str`() utasítással).

Karakterláncok mint listák

A szövegeket a Pythonban szinte minden esetben kezelhetjük úgy, mintha karakterekből álló listák lennének. Lássunk rá egy példaprogramot!

```
1. szöveg = 'szöveg'
2. lista = ['l', 'i', 's', 't', 'a']
3.
4. #kiírás
5. print(szöveg)
6. print(lista)
7. print(''.join(lista))
8.
9. #első és utolsó elem
10. print(szöveg[0], szöveg[-1])
11. print(lista[0], lista[-1])
12.
13. #milyen hosszú a string, hány elemű a lista
14. print(len(szöveg))
15. print(len(lista))
16.
17. #bejárásuk, egy-egy szóközzel, hogy látványosabb legyen
18. for karakter in szöveg:
19.     print(karakter, ' ', end='')
20. print('')
21.
22. for elem in lista:
23.     print(elem, ' ', end='')
24. print('')
```

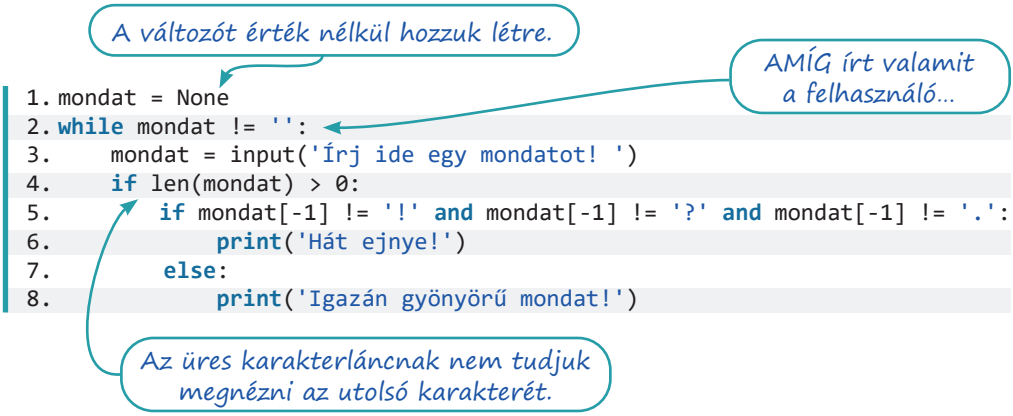
Nincs semmi a két aposztróf között.

Van persze még egy csomó közös tulajdonságuk, és vannak eltérések is – idővel megismerkedünk velük.

Milyen esetben szidja össze az alábbi program a felhasználót?

```
1. mondat = input('Írj ide egy mondatot! ')
2. if mondat[-1] != '!' and mondat[-1] != '?' and mondat[-1] != '.':
3.     print('Hát ejnye!')
4. else:
5.     print('Igazán gyönyörű mondat!')
```

Írjuk át úgy a programunkat, hogy addig kérjen új meg új mondatokat, amíg a felhasználó meg nem elégeli a mókát, és üres bemenetet nem ad (azaz nem ír be semmit, csak lenyomja az Entert)!

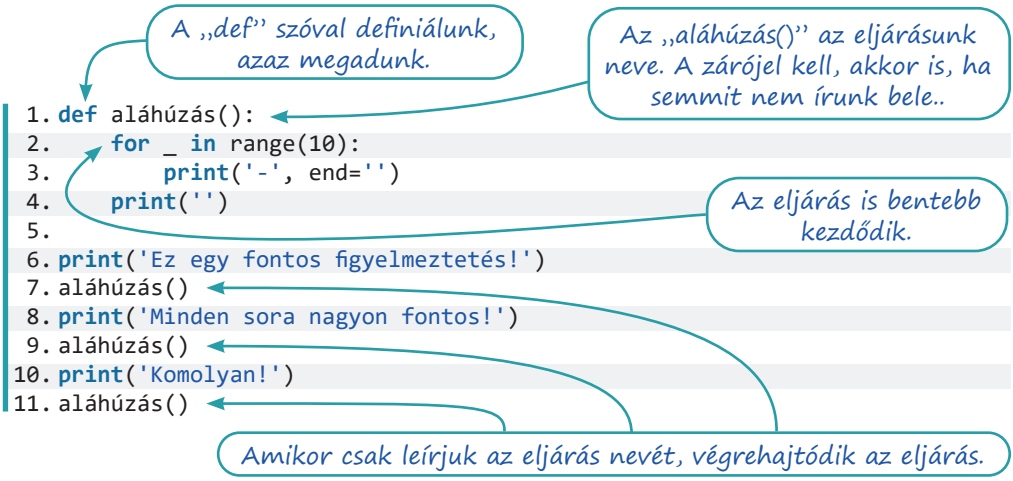



```
1. mondat = None
2. while mondat != '':
3.     mondat = input('Írj ide egy mondatot! ')
4.     if len(mondat) > 0:
5.         if mondat[-1] != '!' and mondat[-1] != '?' and mondat[-1] != '.':
6.             print('Hát ejnye!')
7.         else:
8.             print('Igazán gyönyörű mondat!')
```

A ciklusmag már így is szép nagy, pedig bővíthetnénk is a programunk tudását. Úgyhogy megtanuljuk a programjainkat részekre szedni.

Eljárást írunk

Eljárást két esetben ír a fejlesztő (azaz mi), és mindjárt megbeszéljük, hogy mi ez a két eset. Először azonban meg kell értenünk, hogy mi az eljárás. Vegyünk elő megint egy példaprogramot! A program mindössze annyit tesz, hogy kiír háromsornyi szöveget, aláhúzva. Persze a parancssorban nem tudunk aláhúzott betűket írni, így az „aláhúzás” valójában egy új sor, benne kötőjelekkel. Az eljárás az első négy sorban van.




```
1. def aláhúzás():
2.     for _ in range(10):
3.         print('-', end='')
4.     print('')
5.
6. print('Ez egy fontos figyelmeztetés!')
7. aláhúzás()
8. print('Minden sora nagyon fontos!')
9. aláhúzás()
10. print('Komolyan!')
11. aláhúzás()
```

Amikor a Python azt olvassa a programunkban, hogy `def`, akkor tudja, hogy most figyel, de nem csinál semmit. A `def` után következő részt majd máskor kell végrehajtania. Amikor viszont csak az eljárás nevét olvassa, `def` nélkül (először a 7. sorban látunk ilyet), akkor

1. megkeresi az ilyen nevű eljárást (igen, több eljárás is lehet egy programban),
2. végrehajtja az eljárást,
3. visszajön oda, ahonnan elment végrehajtani az eljárást, és folytatja a programot.

Ha kipróbáltuk a programunkat, akkor megbeszéljük, hogy eljárást két esetben használunk:

- amikor a programunkban több helyen is ugyanaz van, akkor az ismétlődő részt kiszervezzük egy eljárásba,
- de az is lehet, hogy egy helyen van csak egy rész, mégis kitesszük eljárásba, mert úgy könnyebben olvasható a program.

Mire kellenek a zárójelek?

Tegyük fel, hogy többféle aláhúzást is szeretnénk, mondjuk csillagokból állót, meg hullámvonalakból készültet. Akkor most írjunk három eljárást `sima_aláhúzás()`, `csillagos_aláhúzás()` és `hullámos_aláhúzás()` néven? Csak van valami jobb módszer! És tényleg van: a paraméteres eljárás.

Írjuk át az eljárásunkat így:

Ez az eljárás paramétere (vagy argumentuma).

```
1. def aláhúzás(jel):
2.     for _ in range(10):
3.         print(jel, end='')
4.     print('')
```

Itt használjuk fel a paraméter értékét.

Még javítsunk három helyen a programunkban: a hetedik sor legyen ilyen: `aláhúzás('-')`, a kilencedik ilyen: `aláhúzás('*')`, az utolsó meg ilyen: `aláhúzás('~')`. Ha lefuttatjuk a programunkat, látjuk, hogy minden aláhúzás más.

A programunk ilyenkor a következőt csinálja:

1. Amikor az eljárás neve után talál valamit a zárójelben, azt *átadja* az eljárásnak.
2. Kikeresi, hogy az eljárás neve után milyen szó áll (náluk ez volt a `jel`). Ez lesz a paraméter neve, kicsit olyan, mint egy változónév – a továbbiakban ki lehet olvasni az értékét.
3. Az eljárás belsejében ezzel a névvel hivatkozunk az átadott értékre (nálunk a harmadik sorban ezért írja a program azt a karaktert, amit átadtunk az eljárásnak).

Függvényt írunk

A függvény ugyanúgy a program egy elkülönült, magában is értelmezhető része, mint az eljárás. Ugyanabban a két esetben használjuk, mint az eljárást. Ugyanúgy paramétert lehet neki átadni, mint az eljárásnak. Pythonban ugyanazzal a `def` szóval kezdődik a megadása. Akkor mi a különbség? A kimenete pontosan megegyezik. Alaposan hasonlítsuk össze a két kódot!

```
1. def pluszkettő(szám):
2.     print(szám+2)
3.
4. print('5+2= ', end='')
5. pluszkettő(5)
6. print('4+2= ', end='')
7. pluszkettő(4)
```

```
1. def pluszkettő(szám):
2.     return (szám+2)
3.
4. print('5+2= ', pluszkettő(5))
5. print('4+2= ', pluszkettő(4))
```

A bal oldali kódban eljárással valósítjuk meg a feladatot. Az eljárás paraméterében mondjuk meg, hogy melyik számhoz kell kettőt adni. Amikor az eljárást *hívjuk* (azaz leírjuk a nevét, 5. és 7. sor), akkor lefut. Elvégzi az összeadást, és az eredményt kiírja, mert erre utasítja a `print()`. A futás végeztével a kód végrehajtása visszakerül az eljárás hívásának helyére.

A jobb oldalon függvénnyel valósítjuk meg a feladatot. A függvény a hívását követően lefut. Elvégzi az összeadást, visszatér a hívás helyére, az eredményt pedig *visszaadja* a főprogramnak. Olyan, mint ha a függvény futása utáni pillanatban a függvény által visszaadott érték kerülne a függvény nevének helyére, először a 4., majd az 5. sorban. A kiírást a főprogramunk végzi.

Eljárás vagy függvény?

Eljárás és függvény között tehát az a különbség, hogy a függvénynek van visszatérési értéke, az eljárásnak nincs. A visszatérési értéket a függvényben a `return` szót követően adjuk meg.

Az eljárást és a függvényt a legtöbb programozási nyelv nem különbözteti meg élesen, és csak a függvény szót használja. Az ilyen nyelvek – mint a Python is – és az ilyen nyelveken fejlesztők az eljárásokra esetleg csak visszatérési érték nélküli függvényekként tekintenek.

Vissza a kezdetekhez!

Ha újra elővesszük a mondatos programunkat, akkor látjuk, hogy a mondat megítélése egyszerűen kiszervezhető eljárásba. Említettük, hogy eljárásokat két esetben írunk: ha valamit többször kell végrehajtani, vagy pedig olvashatóbbá válik a főprogram. A mondatos programban az utóbbi okból használunk eljárást.

```
 1. def megítél(mondat):
 2.     if len(mondat) > 0:
 3.         if mondat[-1] != '!' and mondat[-1] != '?' and mondat[-1] != '.':
 4.             print('Hát ejnye!')
 5.         else:
 6.             print('Igazán gyönyörű mondat!')
 7.
 8. mondat = None
 9. while mondat != '':
10.     mondat = input('Írj ide egy mondatot! ')
11.     megítél(mondat)
```

Eljárások a gyakorlatban

Egy lényeges megfontolás

Mind az eljárások, mind a függvények megtervezésekor a legtöbbször érdemes úgy dolgoznunk, hogy

- az eljárásunk, illetve függvényünk ne nyúljon a főprogram változóihoz (nemhogy ne próbálja őket átírni, de olvasni se akarja),
- hanem csak a számára híváskor átadott értékekkel dolgozhat.

Vannak programozási nyelvek, ahol erre erősen figyelnünk kell, a Pythonban azonban sokat kell ügyeskedni, ha az eljárás vagy függvény belsejéből babrálni akarunk a főprogram változóival. Szerencsére kifejezetten nem akarunk, úgyhogy ez jó így nekünk.

Feladatok

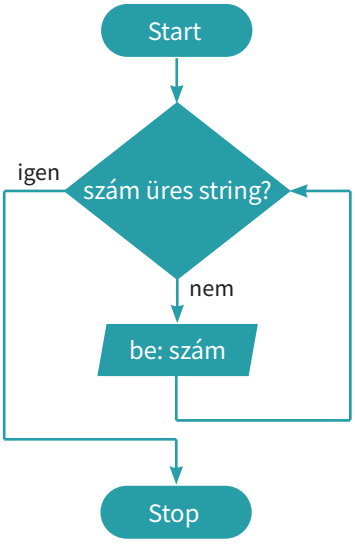
1. Írjuk meg azt az eljárást, amely a számára átadott, literben kifejezett térfogatot átváltja akóba, és az eredményt képernyőre írja! Hogy egy akó hány liter, azt keressük ki az internetről! Sokféle akó van, használjuk a nekünk tetszőt! Hívjuk az eljárást úgy a főprogramból, hogy kiderüljön, hogy 999 liter hány akó!

```
eljárás akóba_vált(liter)
    ki: liter/58,6
eljárás vége

akóba_vált(999)
```

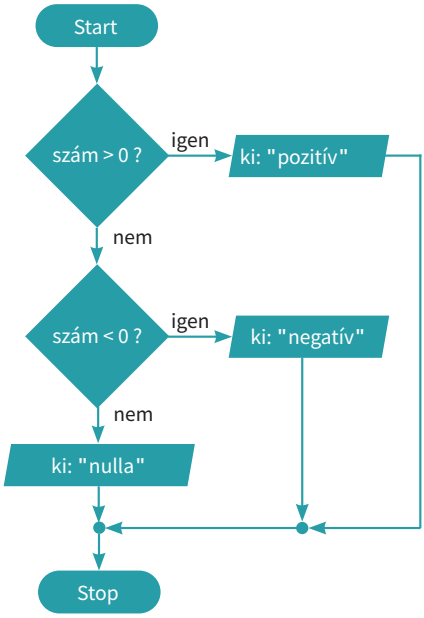
```
1. def akóba_vált(liter):
2.     print(liter/58.6)
3.
4. akóba_vált(999)
```

2. Írjunk olyan programot, amely számokat kér a felhasználótól, amíg üres bemenetet nem kap, majd eljárással kiírja, hogy az épp beírt szám pozitív, negatív vagy nulla!
   a. Először írjuk meg a főprogramot, azaz azt a részt, amely bekéri a számokat (de még ne csináljunk semmit a számmal)!




```
1. szám = None
2. while szám !='':
3.     szám = input('Írj ide egy számot! ')
```

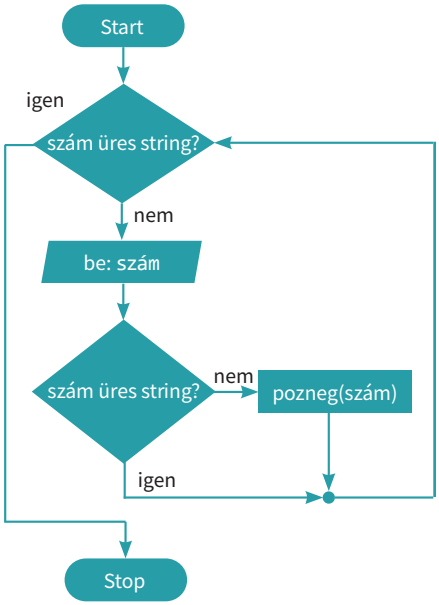
b. A folyamatábra vagy a mondatszerű leírás alapján kódoljuk az eljárást! Az eljárások, függvények nevének megválasztásakor is érdemes figyelnünk arra, hogy a kódot olvasó embert a név segítse a kód megértésében.




```
eljárás pozneg(szám)
    elágazás
    ha szám > 0:
        ki: szám, 'pozitív.'
    különbenha szám < 0:
        ki: szám, 'negatív.'
    különben:
        ki: 'A szám nulla.'
    elágazás vége
eljárás vége
```

```
1. def pozneg(szám):
2.     if szám > 0:
3.         print(szám, 'pozitív.')
4.     elif szám < 0:
5.         print(szám, 'negatív.')
6.     else:
7.         print('A szám nulla.')
```

c. Helyezzük el az eljárást hívó részt a főprogramban! Az eljárás nincs felkészülve rá, hogy üres bemenetet kapjon – ha ilyet kap, a program hibaüzenettel kilép. A főprogramnak figyelnie kell arra, hogy csak akkor hívja az eljárást, ha a bemenet nem üres.




```
1. szám = None
2. while szám != '':
3.     szám = input('Írj ide egy számot! ')
4.     if szám != '':
5.         szám = float(szám)
6.     pozneg(szám)
```

3. Megírandó programunk – igencsak sztereotip döntéseket hozó – bébiszittert szimulál. A program megkérdezi a gyerekek nevét, majd a lányoknak babát, a fiúknak autót ad játszani. A főprogram dolga, hogy neveket kérdezgessen, amíg üres bemenetet nem kap. Egy eljárás dönti el a gyerekek nemét az alapján, hogy a nevük benne van-e a `lánynevek` vagy a `fiúnevek` listában. Ugyanez az eljárás írja ki, hogy az adott gyerek mit kapott játszani. A gyerek nevét az eljárás paramétereként adja át a főprogram.
Azt, hogy valami benne van-e egy listában, nagyon egyszerű eldöntenünk. Mutatunk rá egy példát:

```
1. lista = ['egyik elem', 'másik elem']
2. vizsgált = 'harmadik elem'
3. if vizsgált in lista:
4.     print('Benne van.')
5. else:
6.     print('Nincs benne.')
```

Az „in” ugyanúgy műveleti jel (más szóval: operátor), mint a „+” vagy a „*”.

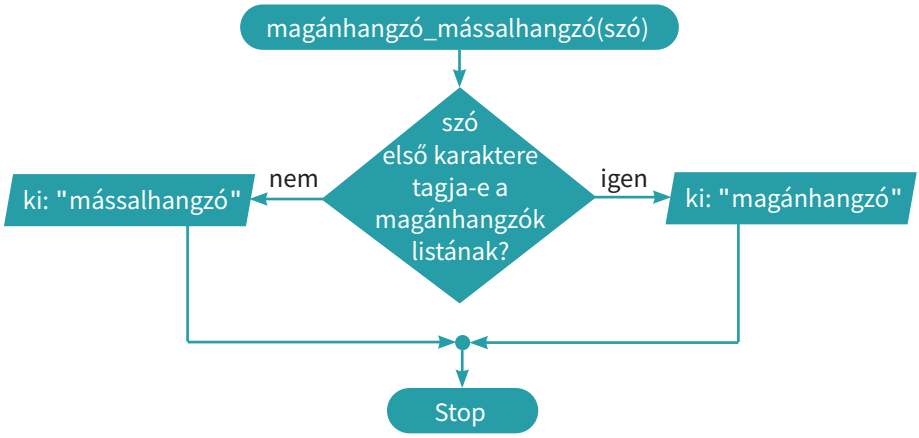
a. Írjuk meg a programot!
b. Egészítsük ki a programot úgy, hogy ha egyik listában sem szerepel a név, akkor kérdezze meg, hogy a gyerek milyen nemű!
c. A `random.choice()` függvény bevetésével oldjuk meg, hogy mind a fiúk, mind a lányok számára több játékból is válogasson a program!

4. Írjunk olyan programot, amelyik listába gyűjti a felhasználó által megadott városokat, majd a listát átadja egy eljárásnak! Az eljárás megszámolja és kiírja, hogy hány európai főváros van az átadott listában.
   a. A program megírását az eljárás megírásával kezdjük, mégpedig azért, mert így sokkal egyszerűbb tesztelni az eljárás működését. Az eljárás mondatszerű leírása:

```
eljárás hány_főváros(városok):
    fővárosok = [’Párizs’, ’Riga’, ’Budapest’, … ]
    fővárosok_száma = 0
    ciklus városok minden város-ára:
        ha város eleme fővárosok-nak:
            fővárosok_száma = fővárosok_száma + 1
    ciklus vége
    ki: fővárosok_száma, ’főváros van a listában.’
eljárás vége
```

   Kódoljuk az eljárást!
   A teszteléskor a főprogram egyetlen sor, például:
   `hány_főváros(['Gárdony', 'Zágráb', 'Nagyhuta', 'Dublin'])`
   b. Gondolkodhatunk fordítva is: a ciklusban a fővárosok lista elemeit járjuk be, és megnézzük, hogy melyik elem található meg az átadott listában.
   A `['Bécs', 'Bécs']` listát vizsgálva melyik megoldás ír válaszul egyet, melyik kettőt?
   c. Írjuk meg a főprogramot! A lista elemeit a `lista_neve.append('új érték')` függvénnyel bővíthetjük. Mielőtt a listát átadnánk az eljárásnak, írjuk ki a lista elemeit egy utasítással! A lista elemeit vesszővel (és szóközzel) válasszuk el! Segít a `join()` függvény.

5. Írjunk olyan eljárást, amely egy paraméterként kapott szóról eldönti, hogy magánhangzóval vagy mássalhangzóval kezdődik, és a döntését képernyőre írja!
   a. Hogyan tudjuk egy változóban tárolt szó első betűjét megkapni?
   b. Meg kell vizsgálnunk, hogy a megkapott első betű benne van-e egy listában. A mássalhangzókat vagy a magánhangzókat érdemes listába gyűjtenünk?




```
eljárás magánhangzó_mássalhangzó(szó)
    magánhangzók = [’a’, ’á’, ’e’, ’é’, ’i’, … ]
    ha szó[0] eleme magánhangzók-nak:
        ki: „magánhangzó”
    különben:
        ki: „mássalhangzó”
eljárás vége
```

```
1. def magánhangzó_mássalhangzó(szó):
2.     magánhangzók = ['a', 'á', 'e', 'é', 'i', 'í', 'o', 'ó',
3.                     'ö', 'ő', 'u', 'ú', 'ü', 'ű']
4.     if szó[0] in magánhangzók:
5.         print(szó, 'magánhangzóval kezdődik.')
6.     else:
7.         print(szó, 'mássalhangzóval kezdődik.')
```

Akármilyen zárójelen belül bármikor kezdhetünk új sort, de írhatjuk egy sorba is.

Függvények a gyakorlatban

A függvények abban különböznek az eljárásoktól, hogy van visszatérési értékük, amit a `return` utasítást követően adunk meg a függvény belsejében. A függvények is írhatnak a képernyőre, de nem szokás ezzel a módszerrel élni. Az iparban a függvényt nagyon gyakran nem ugyanaz a fejlesztő írja, aki majd a programjában használja. Azt szeretjük, ha a függvény fekete doboz: a függvényt használó fejlesztőnek nem kell ismernie a függvény belső működését. Nem tudja, mi történik belül a „dobozban”, csak átadja a függvénynek a feldolgozandó értékeket, a függvény meg visszaadja az eredményt.

Eddig is rengeteg fekete dobozként működő függvényt használtunk – valójában függvény minden olyan „beépített” utasításunk, aminek a neve után zárójel van. Függvény a `len()`, az `int()` és az `str()`, a `join()`, a `random.randint()`. Említettük, hogy a Python az eljárásokra visszatérési érték nélküli függvényként tekint, így például a `print()`-re is, ami annak alapján, amit tanultunk, inkább egy eljárás.

Bár mi egyszerre vagyunk a függvényeink fejlesztői és használói, mégis a fekete doboz elvét szem előtt tartva írjuk meg függvényeinket. Természetesen a függvény a fejlesztése alatt, tesztelési-hibakeresési célból írhat a képernyőre, de a kész függvény már ne tegyen ilyet.

Feladatok

1. Vegyük elő a múltkori akós programunkat, és írjuk át úgy, hogy eljárás helyett függvény legyen benne! Ne felejtkezzünk meg a főprogram módosításáról sem!

```
függvény akóba_vált(liter):
    vissza liter/58,6
függvény vége
ki: „999 liter az”, akóba_vált(999), „akó.”
```

```
1. def akóba_vált(liter):
2.     return liter/58.6
3.
4. print('999 liter az', akóba_vált(999), 'akó.')
```

2. Írjunk olyan függvényt, amely a paraméterként kapott egyjegyű pozitív számot betűkkel leírva adja vissza! Ötlet: a számok neveit írjuk listába, és használjuk a `számok_nevei[szám]` hivatkozást.

```
1. def betűkkel(szám):
2.     számok_nevei = ['nulla', 'egy', 'kettő', 'három', 'négy', …]
3.     return számok_nevei[szám]
4.
5. for szám in range(10):
6.     print(szám, betűkkel(szám))
```

3. Írjunk olyan függvényt, ami a paraméterként kapott szóhoz a megfelelő határozott névelőt adja vissza! A függvény nagyon hasonlít ahhoz az eljárásunkhoz, amelyikkel eldöntöttük, hogy az átadott szó elején magán- vagy mássalhangzó áll.


```
függvény névelő(szó)
    magánhangzók = ['a', 'á', 'e', 'é', 'i', … ]
    ha szó[0] eleme magánhangzók-nak:
        vissza: "az"
    különben:
        vissza: "a"
függvény vége
```

```
1. def névelő(szó):
2.     magánhangzók = ['a', 'á', 'e', 'é', 'i', 'í', 'o', 'ó',
3.                     'ö', 'ő', 'u', 'ú', 'ü', 'ű']
4.     if szó[0] in magánhangzók:
5.         return 'az'
6.     else:
7.         return 'a'
```

Kihívást jelentő feladat: Hogyan tudjuk ezt a függvényt a számok neveit visszaadóval együtt arra használni, hogy a számok elé is a megfelelő névelőt tegye a programunk?

4. Írjunk olyan függvényt, amelynek visszatérési értéke a paraméterként kapott szó hangrendjét adja meg! Szükségünk lesz egy listára a magas, és egy másikra a mély magánhangzókkal. Érdemes lehet logikai változó értékének beállításával jelezni, hogy találtunk-e magas, illetve mély magánhangzót, majd a két változó értéke alapján meghozni a döntést.

```
függvény hangrend(szó)
    mély_magánhangzók = [„a”, „á”, „o”, „ó”, „u”, „ú”]
    magas_magánhangzók = [„e”, „é”, „i”, „í”, „ö”, … ]
    volt_mély := hamis
    volt_magas := hamis
    ciklus szó minden betű-jére:
        elágazás
        ha betű eleme mély_magánhangzók-nak:
            volt_mély := igaz
        különbenha betű eleme magas_magánhangzók-nak:
            volt_magas := igaz
        elágazás vége
```

```
    ciklus vége
    ha volt_mély és nem volt_magas:
        vissza „mély”
    különbenha nem volt_mély és volt_magas:
        vissza „magas”
    különbenha volt_mély és volt_magas:
        vissza „vegyes”
    különben:
        vissza „nem volt magánhangzó a szóban”
függvény vége
```

```
1. def hangrend(szó):
2.     mély_magánhangzók = ['a', 'á', 'o', 'ó', 'u', 'ú']
3.     magas_magánhangzók = ['e', 'é', 'i', 'í', 'ö', 'ő', 'ü', 'ű']
4.     volt_mély = False
5.     volt_magas = False
6.     for betű in szó:
7.         if betű in mély_magánhangzók:
8.             volt_mély = True
9.         elif betű in magas_magánhangzók:
10.            volt_magas = True
11.    if volt_mély and not volt_magas:
12.        return('mély')
13.    elif not volt_mély and volt_magas:
14.        return('magas')
15.    elif volt_mély and volt_magas:
16.        return('vegyes')
17.    else:
18.        return('nincs magánhangzó a szóban')
19.
20. szó = input('Írj ide egy szót! ')
21. print(hangrend(szó))
```

Lehetne úgy is írni, hogy If volt_mély == True and volt_magas == False, de így elegánsabb.

A négyféle visszatérési érték négy return utasítást jelent.

Programunk ebben a formában csak kisbetűs szavakkal működik helyesen. Hogyan tudunk segíteni a problémán?
Kihívást jelentő feladat: Hogyan oldható meg a nagybetűs szavak helyes feldolgozása a magánhangzók listájának bővítése nélkül?

5. Írjunk olyan függvényt, amely egy nevet kapva paraméterként visszaadja a névből képzett monogramot!

```
1. def monogram(név):
2.     szóköz_helye = név.index(' ')
3.     return név[0] + '. ' + név[szóköz_helye+1] + '.'
4.
5. név = input('Írj ide egy nevet! ')
6. print(név, 'monogramja:', monogram(név))
```

Az index() függvény megadja a paraméterének első előfordulását. Listáknál is működik.

Kihívást jelentő feladat: A program jelen formájában például Zsákos Bilbó monogramját hibásan adja meg. Oldjuk meg a problémát! Ötlet:

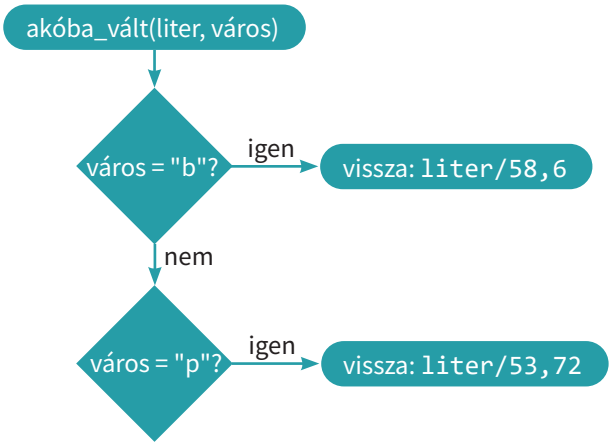
```
if név[0] in ['G', 'L', 'N', 'T'] and név[1] == 'y' …
```

Egy függvénynek több paramétert is átadhatunk. Ilyenkor a paramétereket a függvény neve után álló zárójelben, vesszővel felsorolva adjuk meg. Pont úgy, mint a print() függvény esetében.

```
1. def összeadás(egyik, másik):
2.     return egyik + másik
3.
4. def osztás(osztandó, osztó):
5.     return osztandó/osztó
6.
7. print(összeadás(3, 6))
8. print(összeadás(6, 3))
9.
10. print(osztás(3, 6))
11. print(osztás(6, 3))
```

*A függvények a paraméterek **sorrendjéből** tudják, hogy melyik paramétert melyik változóba tegyék.*

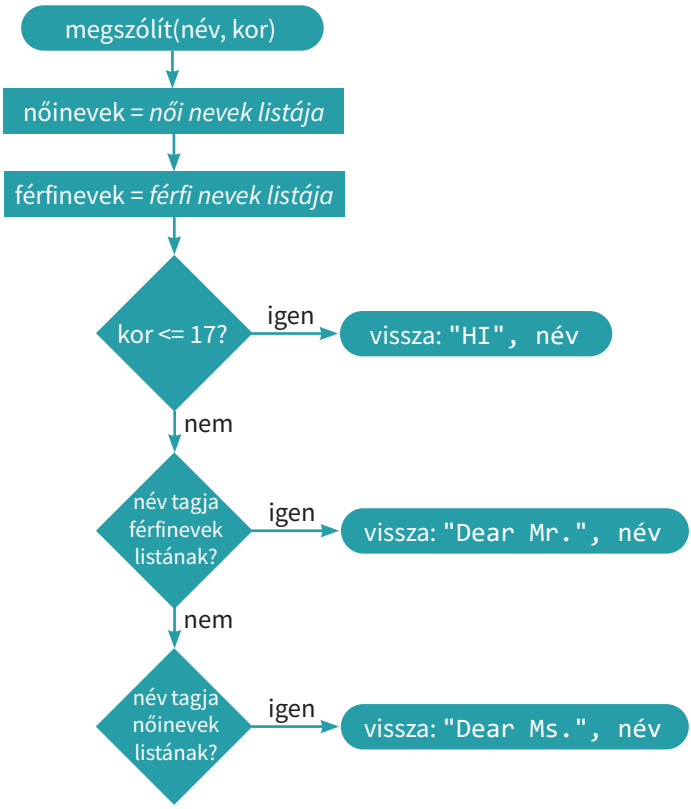
6. Írjuk át az akóátváltós függvényünket úgy, hogy budai és pesti akóba is tudjon váltani! A függvény második paramétere lehet „b” vagy „p”, ez alapján döntse el, hogy mennyivel oszt.




```
függvény akóba_vált(liter, város)
  elágazás
  ha város = „b”:
      vissza: liter/58,6
  különbenha város = „p”:
      vissza: liter/53,72
  elágazás vége
függvény vége
```

```
1. def akóba_vált(liter, város):
2.     if város == 'b':
3.         return liter/58.6
4.     elif város == 'p':
5.         return liter/53.72
6.
7. print('999 liter Budán', akóba_vált(999, 'b'), 'akó.')
8. print('999 liter Pesten', akóba_vált(999, 'p'), 'akó.')
```

7. Írjunk angol megszólítást előállító függvényt! A függvény paramétere egy név és egy évszám. Ha a név Kate, az évszám pedig legfeljebb tizenhét, akkor a megszólítás tegeződős: „Hi Kate”. Ha az évszám nagyobb tizenhétnél, akkor a függvény névlista alapján dönti el, hogy a név férfi vagy női, és a „Dear Mr. Smith” vagy a „Dear Ms. Smith” formát ölti a megszólítás.
   a. Írjuk meg a függvényt úgy, hogy csak egy tagból álló nevet vár!



   b. Nem az első olyan függvényünk ez, amelyik bizonyos esetekben semmit nem ad vissza. Ez kerülendő – adjunk vissza olyan üzenetet, ami elmondja, hogy miért nem adtunk vissza semmi értelmeset!
   c. Módosítsuk úgy a függvényt, hogy két tagból álló nevet kezeljen, a fiatalokat a keresztnevükön, az idősebbeket a vezetéknevükön megszólítva!
   d. Ne csak angolul tudjon a függvényünk! Módosítsuk úgy, hogy harmadik paraméterként egy nyelvet várjon, és a nyelvnek megfelelő megszólítást írjon ki!
   e. Milyen „igazi” alkalmazásban tudunk elképzelni egy ehhez hasonló működésű függvényt?

8. A vogon költészetről az irodalom egyik, mára klasszikussá vált sci-fijéből, a Galaxis útikalauz stopposoknak című műből szerzett tudomást az emberiség. Ha el akarunk benne mélyedni, ám tegyük, de most elég annyit tudni róla, hogy borzasztó. Mi is hasonlóan borzasztó verseket állítunk elő a következő programunkkal.
   A versek sorai mindig egy jelzőből, egy tárgyból, egy állítmányból, valamint mondat végi írásjelből állnak („Nyekergő ablakokat dobálunk!”, „Gömbölyű kútkávát eszünk?!”)

a. Írjunk meg egy olyan, `verssor()` nevű, paraméter nélküli függvényt, amely egy ilyen verssort ad vissza, a mondat négy részét négy listából véletlenszerűen válogatva!
b. Ezt a függvényt hívja egy másik, `versszak()` nevű függvény. A `versszak()` paramétere egy szám, ami megadja, hogy hány sorból álljon a versszak. Ez a függvény a teljes versszakot adja vissza.
c. Az utolsó függvényünk neve: `vers()`. Az első paramétere a versszakok számát megadó szám, a második a verssorok száma versszakonként, visszatérési értéke a teljes vers.
d. A verseink túlcsorduló érzelmekről tesznek tanúbizonyságot, legalábbis az írásjelek alapján. Hogyan tudjuk nagyon egyszerűen elérni, hogy a mondatok végére gyakrabban kerüljön pont?
e. Kihívást jelentő feladat: Írjuk át úgy a `vers()` függvényt, hogy paraméterként mindössze egy listát várjon! Ha a lista [2, 2, 4], akkor olyan verset adjon vissza, amelyik három versszakból áll, és a versszakok rendre 2, 2 és 4 sor hosszúak!

```
1. import random
2.
3. def verssor():
4.     jelzők = ['Nyekergő', 'Gömbölyű', 'Piros', 'Vidám', 'Iszonytató']
5.     tárgyak = ['amőbát', 'ablakokat', 'sárgolyót', 'kútkávát']
6.     állítmányok = ['dobálunk', 'eszünk', 'álmodunk', 'éneklünk']
7.     írásjelek = ['.', '?', '!', '?!']
8.     return random.choice(jelzők) + ' ' + random.choice(tárgyak) + ' ' \
9.         + random.choice(állítmányok) + random.choice(írásjelek)
10.
11. def versszak(sorok_száma):
12.     vsz = ''
13.     for _ in range(sorok_száma):
14.         vsz += verssor() + '\n'
15.     return vsz
16.
17. def egyszerű_vers(versszakok_száma, verssorok_száma):
18.     vers = ''
19.     for _ in range(versszakok_száma):
20.         vers += versszak(verssorok_száma) + '\n'
21.     return vers
22.
23. def jóféle_vers(sorok_számainak_listája):
24.     vers=''
25.     for sorok_száma_a_versszakban in sorok_számainak_listája:
26.         vers += versszak(sorok_száma_a_versszakban) + '\n'
27.     return vers
28.
29. print(egyszerű_vers(3,4))
30. print('---------------')
31. print(jóféle_vers([2,2,4]))
```

A visszaper jelzi, hogy még nincs vége a sornak. Te a gépeden írhatod egy sorba a 8–9. sort. (Ilyenkor nem kell visszaper.)

Variációk típusalgoritmusokra 1.
Történetek a taxisról meg a rókáról

Mik azok a típusalgoritmusok?

Egészen egyszerűen olyan algoritmusok, amelyekkel a programírás során felmerülő problémák egyszerűen megoldhatók. Más néven programozási tételeknek is nevezik őket. Könyvünkben hat egyszerűbb ilyen algoritmussal ismerkedünk meg behatóan, és ezek közül néggyel már találkoztunk is.

A sorozatszámítás – összegzés és átlagolás

1. Egy listában tároljuk, hogy egy taxis egy nap során hány piculát keresett az egyes fuvarjaival. Mennyi pénze lett összesen?

```
bevételek = [1,5,2,3,4]
összes := 0
ciklus bevételek minden bevétel-ére:
    összes := összes + bevétel
ciklus vége
ki: összes
```

```
1. bevételek = [1,5,2,3,4]
2.
3. összes = 0
4. for bevétel in bevételek:
5.     összes += bevétel #vagy: összes = összes + bevétel
6. print('Napi bevétel:', összes, 'picula.')
```

Persze van egyszerűbb megoldás is:

```
1. összes = sum(bevételek)
2. print('Napi bevétel:', összes, 'picula.')
```

Csakhogy ez a rövid megoldás nem minden esetben használható. Például mire megyünk vele a következő feladatban?

2. A róka libát lop a faluból. A libák súlyát – pontosabban tömegét – listában adjuk meg. A farkas a dűlőútnál várja a rókát, és a három kilónál nagyobb libákat elveszi – a piciket nagylelkűen otthagyja a rókának. Hány kiló libát ehet meg a róka?
Az előző feladat algoritmusát felhasználva írjuk meg azt az algoritmust, amelyik ezt a problémát oldja meg!

```
1. libák = [1,5,2,3,4]
2.
3. rókának_jut = 0
4. for liba in libák:
5.     if liba <= 3:
6.         rókának_jut += liba
7. print('A rókának', rókának_jut, 'kilónyi liba marad.')
```

3. Átlagosan hány piculát keres a taxisunk egy fuvarral?
Az előző taxis feladat algoritmusán csak annyit kell módosítanunk, hogy az összeget elosztjuk a lista elemszámával. Ha az első, hosszabb megoldásból indulunk ki, akkor kell még egy változó, ami a ciklus előtt nulla, és a ciklus minden ismétlődésekor eggyel növeljük az értékét. Ez két új sor beszúrása az algoritmusba. Módosítsuk az algoritmust!
A kész kód:

```
1. bevételek = [1,5,2,3,4]
2.
3. összes = 0
4. darab = 0
5. for bevétel in bevételek:
6.     összes += bevétel
7.     darab += 1
8. print('Az átlagos bevétel:', összes/darab, 'picula.')
```

Persze megy ez is egyszerűbben:

```
3. összes = sum(bevételek)
4. darab = len(bevételek)
5. print('Az átlagos bevétel:', összes/darab, 'picula.')
```

Mondjunk olyan átlagolási feladatot, amikor a rövid megoldás nem használható!

4. Átlagosan hány kilósak a rókának maradt libák?

```
1. libák = [1,5,2,3,4]
2.
3. rókának_jut = 0
4. darab = 0
5. for liba in libák:
6.     if liba  <= 3:
7.         rókának_jut += liba
8.         darab += 1
9. print('A róka libái átlagosan', rókának_jut/darab, 'kilósak.')
```

Eldöntés

Megnézzük, hogy van-e adott tulajdonságú elem a listánkban.

5. Volt-e a taxisnak ma ötpiculás bevételű fuvara?
Ezúttal az a feladatunk, hogy addig vizsgálgatjuk ciklussal az értékeket, amíg meg nem találjuk az első olyat, amelyik eleget tesz a feltételnek – ami a mi esetünkben öt.

```
bevételek := [1,5,2,3,4]
vanilyen := hamis
ciklus bevételek minden bevétel-ére:
    ha bevétel = 5:
        vanilyen := igaz
ciklus vége
ha vanilyen:
    ki: „van ilyen”
```

Az algoritmusunk megoldja a problémát, de picit pazarló, nem takarékoskodik a gép erőforrásaival. Végignézi ugyanis az összes értéket a listában, még azt követően is, hogy talált már a feltételnek megfelelőt. Persze ezúttal ez nem gond, de mi van, ha a taxis összes bevétele ott van a listában, mondjuk az előző harminc évről? Szerencsére erre is van megoldás:

```
1. bevételek = [1,5,2,3,4]
2.
3. vanilyen = False
4. for bevétel in bevételek:
5.     if bevétel == 5:
6.         vanilyen = True
7.         break
8. if vanilyen:
9.     print('Van ötpiculás bevétel.')
```

A break arra jó, hogy kiléphessünk a ciklusból, mielőtt az összes elemet bejártuk volna. A többi listalemet úgyis fölösleges volna megvizsgálnunk.

Ahogy már megszoktuk, itt az egyszerűbb megoldás:

```
3. if 5 in bevételek:
4.     print('Van ötpiculás bevétel.')
```

Gondoljunk ki olyan feladatot, ami nem oldható meg a rövid megoldással!

6. Előfordult-e olyan, hogy a róka legalább háromkilós libát lopott?

```
1. libák = [1,5,2,3,4]
2.
3. vanilyen = False
4. for liba in libák:
5.     if liba >= 3:
6.         vanilyen = True
7.         break
8. if vanilyen:
9.     print('Van legalább egy háromkilós liba.')
```

Előfordult-e olyan, hogy a róka kisebb libát hozott, mint az előző napon?
Ebben a feladatban mindenképp index szerint kell bejárnunk a listát, hiszen az aktuális elemet mindig az előzőhöz kell hasonlítani.

```
 1.libák = [1,5,2,3,4]
 2.
 3.vanilyen = False
 4.for index in range(len(libák)):
 5. :   if index > 0:
 6.         if libák[index] < libák[index-1]:
 7.             vanilyen = True
 8.             break
 9.if vanilyen:
10.    print('Volt, amikor kisebb libát lopott az előző napinál.')
```

len(libák) = 5
range(5): 0,1,2,3,4
Az index 0-tól 4-ig fut.

Az első (nulladik sorszámú) elemet még nem vizsgáljuk.

Kiválasztás

Az eldöntéshez képest az a különbség, hogy tudjuk, hogy van adott tulajdonságú elem (ötpiculás bevétel, háromkilósnál nagyobb liba) a listában. De hol van ez az elem? Hányas sorszámú helyen áll?

7. A taxis bevételei közül hányadik volt az ötpiculás?
   Index szerint bejárva a listát végignézzük az egyes elemeket, és megjegyezzük a keresett – és immáron megtalált – elem indexét.

```
bevételek := [1,5,2,3,4]
holvan := speciális_érték
ciklus index := 0-tól bevételek_elemszáma -1 -ig:
    ha bevétel[index] = 5:
        holvan := index
ciklus vége
ki: holvan
```

Észrevehetjük, hogy ez az algoritmus nagyon hasonló az eldöntésnél tárgyalt utolsó esethez – hasonlítsuk össze a két programkódot! Azt is megemlítjük, hogy ez az algoritmus a lehetséges jó válaszok közül mindig csak egyet talál meg. A mondatszerű leírásban az úgynevezett számlálós ciklus szerepel. Az `index` változó értékét változtatja 0 és 4 között. A Python nyelvben ilyen ciklus nincs – helyette bejárósat használunk, a már ismert módon, azaz a bejárandó elemeket a `range()` függvénnyel előállítva. Az index változó ezeket számokat kapja értékül.

```
1. bevételek = [1,5,2,3,4]
2.
3. holvan = None
4. for index in range(len(bevételek)):
5.     if bevételek[index] == 5:
6.     holvan = index
7.     break
8. print('Az ötpiculás fuvar sorszáma:', holvan + 1)
```

A köznyelvben nem nullával kezdjük a számozást – ezért növeljük eggyel a kiírt értéket.

Az egyszerűbb megoldáshoz a lista adattípus index() függvényét használjuk:

```
3.  print('Az ötpiculás fuvar sorszáma:', bevételek.index(5) + 1)
```

8. Megszoktuk már, hogy ilyenkor mindig jön a róka a libáival, és elrontja a jó kis egyszerű megoldás fölött érzett megkönnyebbülésünket. A helyzet most is ez.
   Hányadik napon sikerült a rókának először legalább háromkilós libát lopnia?

```
1. libák = [1,5,2,3,4]
2.
3. holvan = None
4. for index in range(len(libák)):
5.     if libák[index] >= 3:
6.         holvan = index
7.         break
8. print('Az első nagy libát a(z) ', index+1,
9.         '. napon fogta a róka.', sep='')
```

Ezt a két sort írhatjuk egybe is. Ezúttal nem használunk visszapert a sor végén: zárójelen belül nem kell.

Variációk típusalgoritmusokra 2.
Újabb történetek a taxisról meg a rókáról

Keresés

A keresés algoritmusa nem más, mint az eldöntés és a kiválasztás egybeépítése. Az eldöntésnél az a kérdés, hogy van-e olyan elem a listában, amit keresünk, a kiválasztásnál pedig az, hogy melyik ez az elem. Itt mindkét kérdésre válaszolunk.

1. Keressük azt a fuvart, amikor a taxis először keresett öt piculát. Az eldöntés és a kiválasztás algoritmusának ismeretében alkossuk meg az algoritmust, majd kódoljuk.

```
bevételek := [1,5,2,3,4]
vanilyen := hamis
holvan := speciális_érték
ciklus index := 0-tól bevételek_elemszáma -1 -ig:
    ha bevétel[index] = 5:
        vanilyen := igaz
        holvan := index
    elágazás vége
ciklus vége
ha vanilyen:
    ki: holvan
különben:
    ki: „nincs ilyen”
```

Kódolva:

```
1. bevételek = [1,5,2,3,4]
2.
3. vanilyen = False
4. holvan = None
5. for index in range(len(bevételek)):
6.     if bevételek[index] == 5:
7.         vanilyen = True
8.         holvan = index
9.         break
10. if vanilyen:
11.     print('Az ötpiculás fuvar sorszáma:', holvan + 1)
12. else:
13.     print('Nincs ötpiculás fuvar.')
```

Ha találunk megfelelő elemet, mind a két változó értékét beállítjuk.

Ha megszakítjuk a keresést, akkor az első, ha végigmegyünk az összes elemen, akkor pedig az utolsó megfelelő értéket írjuk ki.

Természetesen ezúttal is létezik egyszerűbb megoldás – és ez is a két előző „egyszerűbb megoldás” összeépítése.

```
3. if 5 in bevételek:
4.     print('Az ötpiculás fuvar sorszáma:', bevételek.index(5) + 1)
5. else:
6.     print('Nincs ötpiculás fuvar.')
```

2. Melyik a róka első legalább háromkilós libája?
A rövid megoldás szokás szerint nem használható. A hosszabbat azonban egyetlen karakterben kell *lényegileg* módosítanunk, és máris megoldja ezt a problémát.

Megszámolás

A megszámolás algoritmusával azt derítjük ki, hogy adott tulajdonságú elemből mennyit találunk a listánkban. A lista bejárását végző ciklus előtt létrehozunk egy számláló szerepű változót, és a nulla értéket adjuk neki. Minden egyes találatnál növeljük a számláló értékét.

3. Megszoktuk, hogy a taxis mindig az egyszerű eset: az ő esetében nem csak néhány elem számít a vizsgálatba, mint a rókánál. Ha egy lista *minden* elemét meg kell számolnunk, akkor ugyebár a lista elemszámára, azaz hosszára vagyunk kíváncsiak. Ez pedig a már elég jól ismert `len(bevételek)` utasítással kiderül.
Úgyhogy kivételesen az egyszerű és rövid megoldással kezdjük a sort, és már ezzel is olyan kérdésre válaszolunk, amelyben egy konkrét értéket számolunk meg.
Hány ötpiculás bevétele volt a taxisnak?

```
1. bevételek = [1,5,2,3,4]
2.
3. ennyi = bevételek.count(5)
4. print('Összesen',  ennyi, 'ötpiculás fuvar volt.')
```

Gondoljunk ki olyan feladatot, amelyikre a fenti megoldás nem használható!

4. Talán még emlékszünk rá, hogy a farkas a három kilónál nagyobb libákat veszi el a rókától. Hány libát tarthat meg a róka? Adjunk algoritmust a feladat megoldására, majd készítsük el belőle a kódolt programot!

```
libák := [1,5,2,3,4]
számláló := 0
ciklus libák minden liba-jára:
    ha liba <= 3:
        számláló := számláló + 1
ciklus vége
ki: számláló
```

Kódolva:

```
1. libák = [1,5,2,3,4]
2.
3. számláló = 0
4. for liba in libák:
5.     if liba <= 3:
6.         számláló += 1
7. print(számláló, 'libája marad a rókának.')
```

Csak akkor számoljuk a soron következő libát, ha elég kicsi.

Maximum- és minimumkiválasztás

A kiválasztás tétele arról szól, hogy tudjuk, hogy van adott tulajdonságú elem a listában, de melyik az? Most is tudjuk, hogy van legnagyobb és legkisebb elem, de melyek azok? A maximum és a minimum kiválasztása végezhető együtt is, de azért mi először külön-külön tesszük ezt meg.

5. Melyik volt ma a taxis legjobb fuvarja? Írjuk meg a problémát megoldó algoritmust, majd kódoljuk programmá! Ötlet: vezessünk be egy változót, aminek az értéke a feladatban elképzelhető legkisebb eredmény. Nézzük végig egyesével a listánk elemeit, és ha találunk a változóban tároltnál nagyobb értéket, akkor cseréljük erre a változó tartalmát.

```
bevételek := [1,5,2,3,4]
legtöbb := 0
ciklus bevételek minden bevétel-ére
    ha bevétel > legtöbb:
        legtöbb := bevétel
ciklus vége
```

```
1. bevételek = [1,5,2,3,4]
2.
3. legtöbb = 0
4. for bevétel in bevételek:
5.     if bevétel > legtöbb:
6.         legtöbb = bevétel
7. print('A legjobb fuvar', legtöbb, 'piculát ért.')
```

Ha a mostani bevétel nagyobb, mint az eddigi legnagyobb, akkor mostantól ez a legnagyobb.

Természetesen van egyszerűbb forma:

```
4. print('A legjobb fuvar', max(bevételek), 'piculát ért.')
5. print('A legrosszabb csak', min(bevételek), 'piculát hozott.')
```

Természetesen az egyszerű forma nem mindig elég. Gondoljunk ki olyan feladatot, ahol kevésnek bizonyul!

6. Mekkora a legkisebb liba, amit a farkas elvesz a rókától?

```
1. libák = [1,5,2,3,4]
2.
3. farkas_legkisebb_libája = 100
4. for liba in libák:
5.     if liba >= 4:
6.         if liba < farkas_legkisebb_libája:
7.             farkas_legkisebb_libája = liba
8. print('A farkasnak jutó legkisebb liba',
9.         farkas_legkisebb_libája, 'kg.')
```

A farkas csak a négykilós vagy nehezebb libákkal törődik.

Ha a mostani liba kisebb, mint az eddigi legkisebb, akkor mostantól ez a legkisebb.

Az, hogy a farkas legkisebb libáját tároló változó kezdőértékéül a 100-at választottuk, bátor döntés. Ha ugyanis a róka libáit tartalmazó listában csupa 1-2-3 kilós madár kerül, a végén a programunk azt hazudja majd, hogy a farkas legkisebb libája egy százkilós, strucc méretű liba.

7. Kihívást jelentő feladat: Módosítsuk úgy az előző programunkat, hogy biztosan helyesen adja meg a farkasnak jutó legkisebb liba tömegét!

Fejtörők

Melyik típusalgoritmus való a következő feladatok megoldására? Elég az egyszerűbb forma, vagy kell-e a részletes? Elég érték szerint bejárnunk a listát, vagy index szerint kell?

8. Listában tároljuk, hogy egy londoni pincér mekkora összegeket helyezett el a pénztárcájában az elmúlt órában. Amikor kivett a tárcából pénzt, azt negatív számmal jeleztük. A pincérünk elég ügyetlen és slampos, így soha nem kap borravalót. `[3, 8, 10, 19.35, -6, 5.1, 9, 20]`
   a. Volt-e olyan, hogy a pincér vásárolt valamit, vagy mindig csak neki fizettek?
   b. Ha az óra elején üres a pénztárcája, mennyi van benne az óra végén?
   c. Hány alkalommal kapott biztosan pennyt is, nem csak fontot?
   d. Hány pennyt kapott összesen (feltételezve, hogy csak azt fizették pennyben, amit nem lehet fontban)?
   e. Hány esetben kapott legalább öt fontot?
   f. Mi állt a legnagyobb számlán, amit fizettek a pincérnél?
   g. Ha az óra elején már volt 8 font 23 penny a tárcájában, mennyi pénz volt benne az óra végén?
   h. Hányadik vendég fizetett 9 fontot?
   i. Ha volt olyan vendég, aki tíz fontnál többet fizetett, akkor mondjuk meg, hogy hányadik vendég volt az első ilyen!
   j. Ha volt olyan vendég, aki tíz fontnál többet fizetett, akkor mondjuk meg, hogy hányadik vendég volt az utolsó ilyen!
   k. Volt-e olyan vendég, akinek módjában állt csupa ötfontossal kiegyenlíteni a számlát?
   l. Ha a főnöke minden vendég után fél fontot ad pincérünknek fizetésül, mekkora bevétellel zárta az órát?

9. Tudjuk, hogy a karakterláncok sokszor listaként viselkednek, hosszuk például megállapítható a `len()` függvénnyel. A `'betűK'.isupper()` függvény megmondja, hogy a string minden karaktere nagybetű-e. Ha van egy gyümölcsneveket tartalmazó listánk, akkor az `if 'ló' in gyümölcsök` utasítással megtudjuk, hogy a ló gyümölcs-e.

   A következő feladatok egy mondat szavaiból képzett listára vonatkoznak.

   ```
   ['Én', 'elmentem', 'a', 'vásárba', 'fél', 'pénzen.']
   ```

   a. Hány szóból áll a mondat?
   b. Hány betűs a legrövidebb szó?
   c. Van-e a mondatban olyan szó, ami írásjellel végződik?
   d. Hány névelő van a mondatban, illetve a szavait tartalmazó listában?
   e. Hányadik szó a „fél”?
   f. Van-e a mondatban nagy kezdőbetűs szó, és ha igen, akkor hol?

Variációk típusalgoritmusokra 3.

Feladatok

Ebben a leckében az előző lecke fejtörőit megoldó programokat készítjük el.

Az első feladatcsoportban listában tároljuk, hogy egy londoni pincér mekkora összegeket helyezett el a pénztárcájában az elmúlt órában. Amikor kivett a tárcából pénzt, azt negatív számmal jeleztük. A pincérünk elég ügyetlen és slampos, így soha nem kap borravalót.

A lista: `[3, 8, 10, 19.35, -6, 5.1, 9, 20]`

1. Volt-e olyan, hogy a pincér vásárolt valamit, vagy mindig csak neki fizettek?
   A megoldáshoz az eldöntés típusalgoritmusát használjuk – a részletes algoritmus kell, mert nem egy konkrét érték előfordulását vizsgáljuk.

```
1. bevételek = [3, 8, 10, 19.35, -6, 5.1, 9, 20]
2.
3. vásárolt_a_pincér = False
4.
5. for bevétel in bevételek:
6.     if bevétel < 0:
7.         vásárolt_a_pincér = True
8.         break
9.
10. if vásárolt_a_pincér:
11.     print('Volt, hogy a pincér vásárolt is.')
12. else:
13.     print('Csak a pincérnek fizettek.')
```

Az első két sor a pincéres programjainkban megegyezik, a további példamegoldások a 3. sortól kezdődnek.

2. Ha az óra elején üres a pincér pénztárcája, mennyi pénz van benne az óra végén?
   A megoldáshoz az összegzés típusalgoritmusát használjuk – elegendő az egyszerű forma, mert minden értéket figyelembe kell vennünk.

```
3.  print('A pénztárcában', sum(bevételek), 'font van.')
```

3. Hány alkalommal kapott pennyt is, nem csak fontot?
   A megoldáshoz a megszámolás típusalgoritmusát használjuk – a részletes algoritmus kell, mert nem egy konkrét érték előfordulásait számoljuk meg.
   Akkor kap pennyt is, ha pozitív törtszám a lista vizsgált eleme. Azt, hogy egy szám törtszám-e, kétféleképp is eldönthetjük – mindkét módszer kigondolható az *Elemi adattípusok és elágazások* című lecke utolsó példájában leírtak alapján.

```
 4. ennyiszer_kapott_pennyt = 0
 5.
 6. for bevétel in bevételek:
 7.     if bevétel > 0 and bevétel % 1 != 0:
 8.         ennyiszer_kapott_pennyt += 1
 9.
10. print('A pincér', ennyiszer_kapott_pennyt,
11.         'alkalommal kapott pennyt.')
```

Ha fizetett is pennyvel, azt nem számoljuk, csak azt, amikor kapott.

4. Hány pennyt kapott összesen a pincér?
A megoldáshoz az összegzés típusalgoritmusát használjuk – a részletes algoritmus kell, mert nem minden értéket összegzünk.
Egy angol font száz pennyt ér.

```
 3. ennyi_pennyt_kapott_fontban = 0
 4.
 5. for bevétel in bevételek:
 6.     if bevétel > 0 and bevétel % 1 != 0:
 7.         ennyi_pennyt_kapott_fontban += bevétel % 1
 8.
 9. print('A pincér', ennyi_pennyt_kapott_fontban * 100,
10.         'pennyt kapott.')
```

Csak a bevétel tört részével foglalkozunk.

Alighanem látni fogjuk, hogy a pennyk száma nem egész szám, sok nullát követően egy pici érték még lesz a végén. Ez a probléma a törtszámok kettes számrendszerben való tárolásából adódik. Több módszer is kínálkozik, hogy úrrá legyünk a helyzeten, például:

- az eredmény kerekítése – a `round()` függvénnyel végezhető;
- az eredmény egésszé alakítása – az `int()` függvény használatával;
- a törtszámok nagyobb pontossággal való tárolása – a decimal modul dolga.

5. Hány esetben kapott legalább öt fontot a pincér?
A megoldáshoz a megszámolás típusalgoritmusát használjuk – a részletes algoritmus kell, mert nem egy konkrét érték előfordulásait számoljuk meg.

```
 3. ennyiszer_kapott_legalább_5_fontot = 0
 4.
 5. for bevétel in bevételek:
 6.     if bevétel >= 5:
 7.         ennyiszer_kapott_legalább_5_fontot += bevétel
 8.
 9. print('A pincér',
10.         ennyiszer_kapott_legalább_5_fontot,
11.         'alkalommal kapott legalább 5 fontot.')
```

Írjuk át úgy a programot, hogy ne mindig öt fonthoz viszonyítson – kérdezzük meg a számot a felhasználótól!

6. Mi állt a legnagyobb számlán, amit fizettek a pincérnél?
A megoldáshoz a maximumkiválasztás típusalgoritmusát használjuk – elegendő az egyszerű forma, mert minden értéket figyelembe kell vennünk.

```
3.  print('A legnagyobb számlán', max(bevételek), 'állt.')
```

7. Ha az óra elején már volt 8 font 23 penny a tárcájában, mennyi pénz volt benne az óra végén?
A megoldáshoz az összegzés típusalgoritmusát használjuk – elegendő az egyszerű forma, mert minden értéket figyelembe kell vennünk. A második feladathoz képest annyi a különbség, hogy 8,23-at hozzá kell adnunk a `sum()` függvény eredményéhez.
Írjuk át úgy a programot, hogy egy függvény végezze el a számítást! A függvény paramétere a `bevételek` lista. Ha elkészültünk, módosítsuk úgy a programot, hogy a kezdeti összeget is a felhasználótól kérdezze meg, és ezt is adjuk át paraméterként a számítást végző függvénynek!

A float() függvénnyel alakítjuk az input()-tól kapott szöveget törtszámmá.

A round() függvénnyel kerekítünk két tizedesre.

```
4. def kiszámol(bevételek_listája, kezdeti_pénz):
5.     végső_pénz = sum(bevételek_listája) + kezdeti_pénz
6.     return végső_pénz
7.
8. eleve_a_tárcában = float(input('Mennyi pénz \
9. van a pincérnél az óra elején? '))
10.
11. print('A pénztárcában',
12.         round(kiszámol(bevételek, eleve_a_tárcában), 2),
13.         'font van.')
```

8. Hányadik vendég fizetett 9 fontot?
A megoldáshoz a keresés típusalgoritmusát használjuk – elegendő az egyszerű forma, mert minden értéket figyelembe kell vennünk, és egy konkrét értéket keresünk.

```
3.  print('A ', bevételek.index(9)+1,
4.          '. vendég fizetett 9 fontot.', sep='')
```

9. Ha volt olyan vendég, aki tíz fontnál többet fizetett, akkor mondjuk meg, hogy hányadik vendég volt az első ilyen!
A megoldáshoz a keresés típusalgoritmusát használjuk – a részletes algoritmus kell, mert nem egy konkrét érték előfordulását keressük. A listát kénytelenek leszünk index szerint bejárni, mert szükségünk lesz a megtalált érték listában elfoglalt helyére is. Ha nagyon ódzkodunk az index szerinti bejárástól, akkor az is jó megoldás, hogy megtaláljuk az első tíznél nagyobb értéket, majd a 8. feladatnál is látható `index()` függvénnyel megkérdezzük, hogy hányadik helyen áll.

```
van_több_mint_10 = False
hol_van = 0
for index in range(len(bevételek)):
    if bevételek[index] > 10:
        van_több_mint_10 = True
        hol_van = index
        break

if van_több_mint_10:
    print('Az első tíz fontnál többet fizető vendég a(z) ',
            hol_van+1, '. vendég.', sep='')
```

10. Ha volt olyan vendég, aki tíz fontnál többet fizetett, akkor mondjuk meg, hogy hányadik vendég volt az utolsó ilyen!
A megoldás nagyon hasonló az előző feladatéhoz. Az utolsó vendég fellelésére két lehetséges módszer:

- Kihagyjuk a `break` utasítást – ilyenkor végiglépdel a ciklusunk az összes listaelemen, és az utolsó tíznél nagyobb elem pozíciója marad a `hol_van` változóban. Ez hosszú listáknál pocséklás, azaz a megoldás működik, de nem hatékony.
- A bejárandó indexeket előállító `range()` függvényt így paraméterezzük: `range(len(bevételek)-1, -1, -1)`. A `len()` 8-at ad vissza, ezért az első –1 használatával kivonunk belőle egyet, hogy a lista utolsó értékétől, azaz héttől járjuk be a listát. A második –1 az első olyan értéket adja meg, amit a `range()` már nem ad vissza – azaz 0-ig tart a bejárás. A harmadik –1 azt mondja meg, hogy egyesével akarunk haladni visszafelé. Ez a korrekt megoldás. Sok más nyelven ezt egy visszafelé haladó számlálós ciklussal valósítanánk meg – de Pythonban nincs ilyen.

11. Írjuk ki, ha volt olyan vendég, akinek módjában állt csupa ötfontossal kiegyenlíteni a számlát!
A megoldáshoz az eldöntés típusalgoritmusát használjuk – a részletes algoritmus kell, mert nem egy konkrét érték előfordulását vizsgáljuk. Csak akkor kell kiírnunk valamit, ha találunk megfelelő értéket, ezért picit egyszerűsítünk a megoldáson.

```
for bevétel in bevételek:
    if bevétel % 5 == 0:
        print('Volt olyan vendég, aki tudott csupa ötössel fizetni.')
        break
```

12. Ha a főnöke minden vendég után fél fontot ad pincérünknek fizetésül, mekkora bevétellel zárta az órát?
A megoldáshoz a megszámolás típusalgoritmusát használjuk – a részletes algoritmus kell, mert nem minden értéket akarunk megszámolni, és nem egy konkrét érték előfordulásainak számát.

```
3. vendégek = 0
4.
5. for bevétel in bevételek:
6.     if bevétel > 0:
7.         vendégek += 1
8.
9. print('A pincér', round(vendégek/2, 2 ), 'font fizetést kap.')
```

A második feladatcsoport feladatai a következő, egy mondat szavait tartalmazó listára vonatkoznak:

```
['Én', 'elmentem', 'a', 'vásárba', 'fél', 'pénzen.']
```

13. Hány szóból áll a mondat?
A megoldáshoz a megszámolás típusalgoritmusát használjuk – elegendő az egyszerű forma, mert minden értéket figyelembe kell vennünk.

```
1. mondat = ['Én', 'elmentem', 'a', 'vásárba', 'fél', 'pénzen.']
2.
3. print('A mondat', len(mondat), 'szóból áll.')
```

Az első két sor a további programjainkban megegyezik, a következő példamegoldások a 3. sortól kezdődnek.

14. Hány betűs a legrövidebb szó?
A megoldáshoz a minimumkiválasztás típusalgoritmusát használjuk – a részletes algoritmus kell, mert nem a lista értékei, hanem az azokból képzett számok között keresünk.

```
3. legrövidebb_szó_hossza = 1000
4.
5. for szó in mondat:
6.     if len(szó) < legrövidebb_szó_hossza:
7.         legrövidebb_szó_hossza = len(szó)
8.
9. print('A legrövidebb szó', legrövidebb_szó_hossza, 'karakteres.')
```

Olyan kezdeti értéket választunk, aminél biztosan van kisebb.

Ha a mostani szó rövidebb, mint az eddigi legrövidebb, akkor mostantól ez a legrövidebb.

Kis gond van a programunkkal: a szó hosszába beleszámítja az írásjeleket is. Ezen most nem fogunk segíteni.

15. Írjuk ki, ha van a mondatban olyan szó, ami után mondatvégi írásjel áll!
A megoldáshoz az eldöntés típusalgoritmusát használjuk, ráadásul kétszer is. A fő eldöntéshez a részletes algoritmus kell, mert nem közvetlenül a lista értékeit vizsgáljuk. Az egyes szavak végének vizsgálatakor viszont jól jön az egyszerű forma. Vegyük észre, hogy az írásjeleket tartalmazó karakterláncban kereshetünk lista módjára – bár használhatnánk listát itt is.

```
3.  írásjelek = '.?!'
4.
5.  for szó in mondat:
6.      if szó[-1] in írásjelek:
7.          print('Van olyan szó, ami után írásjel áll.')
```

A –1. index az utolsó karaktert jelenti.

16. Hány névelő van a mondatban?
A megoldás nagyon hasonló az előző feladatéhoz – ezúttal nem csak az utolsó karaktert hasonlítjuk, hanem az egész szót, és az írásjelek helyett a névelők listájára van szükség.

17. Hányadik szó a „fél”?
A megoldáshoz a keresés típusalgoritmusát használjuk – elegendő az egyszerű forma, mert minden értéket figyelembe kell vennünk, és teljes szót keresünk.

```
3.  print('A mondatban a "fél" szó a', mondat.index('fél')+1,'. helyen áll.', sep='')
```

18. Van-e a mondatban nagy kezdőbetűs szó, és ha igen, akkor hol?
A megoldáshoz a keresés típusalgoritmusát használjuk – a részletes algoritmus kell, mert nem a lista értékei, hanem az azokból képzett értékek figyelésével hozunk döntést. A listát index szerint járjuk be, mert azt is tudni akarjuk, hogy hányadik szó kezdődik nagy kezdőbetűvel. Az `isupper()` függvénnyel vizsgáljuk, hogy egy adott betű nagybetű-e.

```
3.  van_nagy_kezdőbetűs = False
4.  hol_van = None
5.  for index in range(len(mondat)):
6.      if mondat[index][0].isupper():
7.          van_nagy_kezdőbetűs = True
8.          hol_van = index
9.
10. if van_nagy_kezdőbetűs:
11.     print('A(z) ', hol_van+1,
12.             '. szó kezdődik nagybetűvel.', sep='')
```

A mondat indexedik szavának nulladik sorszámú karaktere.

A fenti kód sem hatékony. Miért? Hogyan oldható meg, hogy amikor már találtunk nagybetűs szót, hagyjuk is abba a keresést?
Módosítsuk úgy a programot, hogy a keresést végző rész kerüljön függvénybe! A függvény paramétere a mondat szavait tartalmazó lista, a függvény visszatérési értéke pedig egy lista, a nagybetűs szavak indexeivel.

Listákat tartalmazó listák – kétdimenziós adatszerkezet

Mik azok a kétdimenziós adatszerkezetek?

Az egyszerű listák egydimenziós adatszerkezetek – azaz csak hosszuk van, mint egy szakasznak a geometriában. Azonban a listákban elhelyezhetünk olyan elemeket is, amelyek saját maguk is listák. Így lesz az adatszerkezet kétdimenziós: van „szélessége” és „magassága.”

A következő lista egy vonósnégyes tagjainak jelenléti íve. Az ív egy hét munkanapjain mutatja a jelenlétet, ahol 1 szerepel benne, ott jelen volt a zenész, ahol 0, ott nem. Egy-egy „kis lista” egy zenész jelenlétét mutatja be, a „nagy” lista pedig az egész zenekarét.

```
1.  ív = [[1, 1, 1, 1, 1],
2.       [1, 1, 1, 1, 0],
3.       [1, 1, 0, 0, 0],
4.       [0, 1, 1, 1, 1]]
5.
```

egyik hegedűs

másik hegedűs

brácsás

csellós

Figyeljük meg, hogy a kis listák között vessző van – hiszen most a kis listák a nagy lista elemei, márpedig egy lista elemeit vesszővel soroljuk fel. Látjuk, hogy a csellós hétfőn hiányzik, péntekre pedig kiről a második hegedűs és a brácsás. Hogy lesz így előadás szombaton?!

A kétdimenziós adatszerkezetek használata nagyon gyakori – pont úgy, ahogy a táblázatoké a valóságban.

Listákat tartalmazó listák bejárása

Amikor ezt a kétdimenziós listát bejárjuk, a ciklusváltozóba mindig egy-egy vonós egész heti jelenléti íve, azaz egy kis lista kerül. Próbáljuk ki!

```
6.  for vonós in ív:
7.      print(vonós)
```

Ha az egyes vonósok jelenléti ívét tartalmazó kis listákat is be akarjuk járni, akkor egymásba ágyazott ciklusokat kell írnunk:

```
6.  for vonós in ív:
7.      for nap in vonós:
8.          if nap == 1:
9.              print('itt', ' ', sep='', end='')
10.         else:
11.             print('otthon', ' ', sep='', end='')
12.     print()
```

Ez a print() az egyes zenészek sorai végén új sort kezd.

Típusalgoritmusok a kétdimenziós adatszerkezetekben

Az alábbi példákban a zenészes listánkkal dolgozunk. A példakódok mindig a hatodik sortól indulnak, mert az első négy sor (meg utána egy üres sor) csak a jelenléti ív adatait tartalmazza – a korábban ismertetett módon.

1. A zenészek a közeli kifőzdében szoktak ebédelni. Ismerik őket, úgyhogy hét közben csak felírják a fogyasztott adagok számát, és péntekenként fizetik az egész heti számlát. Hány adagot fizetnek ezen a pénteken?
   Az első megoldásban egyszerűen megszámoljuk, hogy hány egyes van a „táblázatban”:

```
adagok = 0
for vonós in ív:
    for nap in vonós:
        if nap == 1:
            adagok += 1  #vagy adagok = adagok + 1
print('Összesen', adagok, 'adagot kell fizetni pénteken.')
```

   A második megoldásban egy vonós heti fogyasztását a `sum()` függvénnyel adjuk meg. Ez a függvény összeadja az átadott listában vagy más bejárható objektumban lévő számokat: `sum([4,3,2]) = 9; sum(range(4)) = 6.`

```
adagok = 0
for vonós in ív:
    adagok += sum(vonós)
print('Összesen', adagok, 'adagot kell fizetni pénteken.')
```

   összegzés egyszerű formája egy sorban

   Az előző megoldásban kihasználtuk, hogy történetesen egyesekkel jelezték a jelenlétet. Ha például a ’jelen volt’ karaktersorozattal jelezték volna, a megoldásunk fabatkát sem érne. Ilyenkor nem összeadnunk, hanem megszámolnunk kell, amire a megszámolás típusalgoritmusának egyszerű formája, a `count()` függvény alkalmas. Ezt a lista után írjuk, egy ponttal a listához kötve. A függvény egyébiránt karakterláncokkal, azaz stringekkel is működik: `'rendetteremtettem'.count('e') = 6.`

```
adagok = 0
for vonós in ív:
    adagok += vonós.count(1)
print('Összesen', adagok, 'adagot kell fizetni pénteken.')
```

   megszámolás egyszerű formája egy sorban

   A Python nem volna Python, ha nem lehetne egyetlen sorban is elegánsan megoldani ezt a feladatot – akit érdekel, keressen a *list comprehension* (azaz listaértelmezés) és a *flattening* (azaz lapítás) kifejezésre az interneten!

2. Melyik zenész volt a legtöbbet jelen a héten? Melyik a legkevesebbet?
   Erre a kérdésre jobb híján a zenész sorszámával válaszolunk. A sok lehetséges megoldásban két nagyobb csoportot különítünk el.
   Az elsőben egyetlen ciklus lefuttatásával kapjuk meg a megoldást. A ciklus lényegében egy maximumkiválasztás. Ha arra gondolunk, hogy az összes érték között keresünk, és az értékeket nem manipuláljuk – nem a négyzetük vagy köbgyökük között keressük a legnagyob-

bat –, akár eszünkbe juthatna az algoritmus egyszerűbb formáját is használni. Aztán eszünkbe jut, hogy ez azért nem lesz jó, mert *maguk az értékek nem állnak rendelkezésre azonnal feldolgozható formában*. Így a ciklusban először előállítjuk az értékeket, és párhuzamosan vizsgáljuk, hogy az épp soron következő zenész többször volt-e jelen, mint az eddigi legtöbbet jelen lévő.

```
 6. sorszám = None
 7. legnagyobb = 0
 8.
 9. for index in range(len(ív)):
10.     vonós_jelenléte = sum(ív[index])
11.     if vonós_jelenléte > legnagyobb:
12.         legnagyobb = vonós_jelenléte
13.         sorszám = index
14. print('A(z) ', sorszám+1, '. vonós volt a legtöbbet jelen.', sep='')
```

összegzés egyszerű formája a maximumkiválasztás kellős közepén

A második megoldáscsoport úgy fog hozzá a probléma megoldásához, hogy egy ciklussal először előállítja azt a listát, amiben keresgélni kell. Ha ezt a listát előállítottuk, megkeressük a legnagyobb értéket (maximumkiválasztás), majd megtudjuk az érték pozícióját (kiválasztás).

```
 6. vonósok_jelenlétei = []
 7.
 8. for vonós in ív:
 9.     vonós_jelenléte = sum(vonós)
10.     vonósok_jelenlétei.append(vonós_jelenléte)
11.
12. sorszám = vonósok_jelenlétei.index(max(vonósok_jelenlétei))
13.
14. print('A(z) ', sorszám+1, '. vonós volt a legtöbbet jelen.', sep='')
```

Ez a ciklus tölti fel az új listát az append() függvényt használva.

maximumkiválasztás

keresés

3. Volt-e olyan zenész, aki mindig jelen volt?
A feladat az előzőnél egyszerűbb: nem kell maximumot keresni, tudjuk, hogy azt kell eldöntenünk, hogy volt-e valaki ötször.

4. Írjuk ki, ha volt olyan nap, amikor mindenki jelen volt a próbán!
A feladat nehézsége, hogy ezúttal nem egy vonóst, hanem egy napot vizsgálunk. Úgy is mondhatjuk, hogy nem egy sor értékeit összegezzük, hanem egy oszlopét.

```
 6. for nap_indexe in range(5):
 7.     e_napon_ennyien_voltak = 0
 8.     for vonós in ív:
 9.         e_napon_ennyien_voltak += vonós[nap_indexe]
10.     if e_napon_ennyien_voltak == 4:
11.         print('A(z) ', nap_indexe+1,
12.               '. napon mindenki jelen volt.', sep='')
```

Összegzés: 0 vagy 1 hozzáadása. Minden „napon” lefut.

eldöntés

Objektumok adatai kétdimenziós listákban

Programozásórán még nem hangsúlyoztuk, de épp ideje szokni azt a gondolatot, hogy a programjaink *objektum*okkal – árucikkekkel, tanulókkal, órarendekkel, menetrendekkel, könyvekkel és még ki tudja mivel – dolgoznak. Végső soron objektumok szerepeltek az előző, jelenléti íves példában is, de ezúttal jelenlétük nyilvánvalóbb lesz. Egy osztály tanulóinak adatait tároljuk egy 2D-listában. Egy-egy kis lista egy-egy objektumról, azaz egy-egy diákról szól. A kis lista elemei, azaz az objektumainkról tárolt tulajdonságok: név, nem, kor és e-mail-cím. Íme:

```
1.  osztály = [['Noémi', 'l', 15, 'noemi@hipp.hopp'],
2.             ['Dezső', 'f', 17, 'dezso2@nyikk.nyekk'],
3.             ['Gizella', 'l', 16, 'gizi@pikk.pakk'],
4.             ['Edömér', 'f', 16, 'edo@itt.ott']]
5.
```

A példakódok a már ismert módszer szerint a hatodik sornál kezdődnek.

1. Írjuk ki az osztály névsorát! Írjuk ki minden név mellé az e-mail-címet is!

```
6.  for tag in osztály:
7.      print(tag[0], ": ", tag[-1], sep='')
```

4. Mennyi az osztály átlagéletkora?

```
6.  összeg = 0
7.  for tag in osztály:
8.      összeg += tag[2]
9.
10. print('Az osztály átlagéletkora', összeg/len(osztály), 'év.')
```

9. A lányok vannak többen, vagy a fiúk?

```
6.  lányok = 0
7.  for tag in osztály:
8.      if tag[1] == 'l':          <-- kis L betű
9.          lányok += 1
10.
11. if lányok > len(osztály) - lányok:
12.     print('A lányok vannak többen.')
13. elif lányok < len(osztály) - lányok:
14.     print('A fiúk vannak többen.')
15. else:
16.     print('A fiúk pont annyian vannak, mint a lányok.')
```

10. Kérjünk be egy nevet a felhasználótól, és válaszoljunk a megfelelő ember e-mail-címével! Ötlet: tároljuk a nevet egy változóban, majd járjuk be a listát, és ahol `tag[0]` a megadott név, ott írjuk ki `tag[-1]`-et! Kell-e `break` a kiírást követően?

Objektumok szótárban

A szótár adattípus

Eddig egyetlen összetett adattípusunk van, mégpedig a lista. (Illetve van még egy, az úgynevezett range típus, amit a `range()` függvénnyel állítunk elő, de önmagában szinte semmire nem használjuk.) Ebben a leckében megismerkedünk a szótár adattípussal. Előrebocsátjuk, hogy a szótár adattípus nem *szükséges* abban az értelemben, hogy minden, amire a szótár típus használható, megoldható listák használatával is, de a szótárak használata olyan sok esetben teszi kényelmesebbé a munkánkat, hogy kifejezetten *érdemes* megismerkednünk vele.

A megismerkedéshez írjunk egy egyszerű magyar–angol szótárat. (Senki ne gondolja azonban, hogy a szótár adattípussal csak a köznapi értelemben vett szótárat lehet készíteni!) A szótárunkban tárolunk néhány szót, és a programunknak képesnek kell lenni arra, hogy amíg a felhasználó üres bemenetet nem ad, addig megkeresi neki a magyar szó angol megfelelőjét. A szótárunkat első közelítésben még a szótár adattípus használata nélkül, pusztán a lista adattípus használatával valósítjuk meg.

```
1.  def szótáraz(magyar_szó):
2.      szótár = [['nagy', 'big'], ['pici', 'tiny'],
3.                ['én', 'I'], ['foci', 'soccer'],
4.                ['ott', 'there'], ['szarvas', 'deer'],
5.                ['soha', 'never']]
6.
7.      for szó in szótár:
8.          if szó[0] == magyar_szó:
9.              return szó[1]
10.
11.     return 'Nincs ilyen szó a szótárban.'
12.
13.
14. kérdezett_szó = None
15.
16. while kérdezett_szó != '':
17.     kérdezett_szó = input('Melyik magyar szóra vagy kíváncsi? ')
18.     if kérdezett_szó != '':
19.         print(kérdezett_szó, 'keresésének eredménye:',
20.               szótáraz(kérdezett_szó))
```

Minden szópár egy kis lista. A gépeden írhatod egy sorba.

keresés

Itt kezdődik a főprogram.

Itt hívjuk a függvényt.

Ez a szokásos kérdezős ciklusunk – addig kérdez, amíg üres bemenetet nem kap.

A szótárazás pont olyan feladat, hogy már érdemes kiszerveznünk önálló függvénybe, hogy a kódunk könnyen olvasható maradjon. A feladatot egy ciklus végzi, amiben a keresés algoritmusát alkalmasan átalakítottuk. Nem tartjuk nyilván, hogy volt-e találat, hiszen, ha volt, a 9. sorban lévő `return` úgyis megszakítja a függvény futását, és visszatér a találattal.

A második közelítésben már bevetjük a szótár adattípust, és menet közben meg is ismerkedünk vele. Aki lépésről lépésre hasonlítja össze az új programot az előzővel, annak szólunk, hogy itt nem lesz függvényünk. A szavakat tároló adatszerkezetünk, azaz egy szótár nevű és szótár típusú szerkezet így néz ki:

```
1. szótár = {'nagy': 'big', 'pici': 'tiny',
2.           'én': 'I', 'foci': 'soccer',
3.           'ott': 'there', 'szarvas': 'deer',
4.           'soha': 'never'}
```

Figyeljük meg, hogy nincsenek „kis listák". **A szótár egy megfeleltetési adattípus, amiben kulcs-érték párokat helyezünk el.** Egy ilyen kulcs-érték pár a `'nagy':'big'`. A *nagy* a kulcs, a neki megfelelő érték a *big*. Az ilyen kulcs-érték párosok felsorolása a szótár, amit – megkülönböztetendő a listától – kapcsos zárójelekben adunk meg.

A szótárak és a kétdimenziós listák használata közötti leglényegesebb különbség, hogy egy szó kikeresése ezúttal csak ennyi: `print(szótár['nagy'])`. Írjuk csak be ötödik sornak, és próbáljuk ki! Nincs ciklus, nem használjuk a keresés típusalgoritmusát. Ha már látjuk, hogy ez milyen kényelmes, valósítsuk meg a szótárprogramunkat a szótár használatával!

Az 5. sor üres, ezért nem írjuk ide.

```
6. kérdezett_szó = None
7.
8. while kérdezett_szó != '':
9.     kérdezett_szó = input('Melyik magyar szóra vagy kíváncsi? ')
10.    if kérdezett_szó != '':
11.        print(kérdezett_szó, 'keresésének eredménye:',
12.              szótár[kérdezett_szó])
```

Ha alaposan teszteltük a programunkat, bizonyára feltűnt, hogy nem törődtünk azzal az esettel, amikor a szótárban nincs meg a keresett érték. Ráadásul a programunk el is halálozik, ha nem létező értéket kerestetünk vele. A helyzetet megoldja a szótár adattípus `get()` függvénye. Cseréljük le a 12. sort erre!

```
12.              szótár.get(kérdezett_szó, 'Nincs ilyen szó.'))
```

A szótáraknak azonban csak a kulcsai között tudunk ilyen egyszerűen keresni. Ha a magyar–angol szótárunkat angol–magyarrá alakítanánk, a listás megoldásnál egyszerűen `szó[0]` helyett `szó[1]`-et írunk, és fordítva. A szótáras megoldásnál viszont ilyenkor már tényleg be kell járnunk a szótárat. A bejárás alapesetben a kulcsokat teszi a ciklusváltozóba: `for kulcs in szótár`. Ha az érték is kell, akkor a ciklus belsejében használhatjuk a `szótár[kulcs]` formát, de pythonosabb, ha egyszerre kérjük el a kulcsot és az értéket is a `for kulcs, érték in szótár.items()` sorral. Próbáljuk ki a szótárunk összes értékének kiírását mindkét módszerrel!

Megjegyezzük még, hogy

- egy szótár kulcsai között sosem lehet két azonos.
- a szótár kulcsait visszakapjuk a `szótár.keys()`, az értékeket a `szótár.values()` függvénnyel – és ezek is bejárhatók, valamint használható velük az `in` operátor vagy a `sum()` függvény.
- a szótárak értékei maguk is lehetnek listák vagy szótárak, és a szótárak is tehetők listába.
- létező szótárba a `szótár['új kulcs'] = 'új érték'` utasítással vehető fel új érték, a szükségtelenné vált kulcs-érték pár törlésére pedig a `del szótár['törlendő kulcs']` utasítás való.

Osztálytagok a szótárban

A szótárakat nagyon gyakran használjuk arra, hogy egy-egy objektum tulajdonságait tároljuk egy szótárban. Az előző lecke utolsó feladatában, ahol az osztálytagok adatait – tulajdonságait – tároltuk, a tárolást listával oldottuk meg. Ott, ha például valakinek a korára voltunk kíváncsiak, azt nekünk kellett tudni, hogy a kis lista 0. elemeit figyelve keresünk, majd a 2. elemet visszaadva kapjuk meg a választ. Ebben változtat nagyot a szótár, hiszen ha ügyesen alkotjuk meg, elég majd ennyit mondanunk: `print(osztály['Dezső'][kor])`. Ez kényelmesebb megoldás, de nem ez a lényeg, hanem az, hogy első ránézésre látjuk, hogy mit csinál a kód. Manapság ugyanis egy szoftverfejlesztő sokkal gyakrabban foglalkozik egy régi, esetleg nem is őáltala írt kód karbantartásával, javítgatásával, mint új programok írásával.

Ahogy a kétdimenziós listánkat szótárrá alakítjuk, talán ez lesz az első megálló (az e-mail-címek tárolásáról lemondunk, mert a könyvben túl hosszúak lennének a sorok, és áttekinthetetlen lenne a szerkezet):

```
1. osztály = [{'név': 'Noémi', 'nem': 'l', 'kor': 15},
2.            {'név': 'Dezső', 'nem': 'f', 'kor': 17},
3.            {'név': 'Gizella', 'nem': 'l', 'kor': 16},
4.            {'név': 'Edömér', 'nem': 'f', 'kor': 16}]
```

Nos, ez még csak egy szótárakat tartalmazó lista. Ha Dezső korára vagyunk kíváncsiak, a programunk így néz ki:

```
6. for tag in osztály:
7.     if tag['név'] == 'Dezső':
8.         print(tag['kor'])
```

Azaz tudunk már név szerint hivatkozni az egyes tulajdonságokra, és ez jó. Továbbra is be kell járnunk a listát, ez nem jó. Legyünk bátrabbak, és legyen az egész lista egy olyan szótár, amiben a nevek a kulcsok, az értékek pedig az egyes osztálytagok többi tulajdonságát tároló kisebb szótárak! (Figyelem, ezzel a megoldással lehetetlenné tesszük két azonos nevű osztálytag tárolását!)

```
1. osztály = {'Noémi': {'nem': 'l', 'kor': 15},
2.            'Dezső': {'nem': 'f', 'kor': 17},
3.            'Gizella': {'nem': 'l', 'kor': 16},
4.            'Edömér': {'nem': 'f', 'kor': 16}}
5.
6. print(osztály['Dezső']['kor'])
```

Ezért küzdöttünk!

Ha elővesszük az osztályhoz kapcsolódó többi feladatot az előző leckéből, látjuk, hogy mindegyiket meg tudjuk szótárral is oldani.

1. Mennyi az osztály átlagéletkora?

```
6. összeg = 0
7. for tag in osztály:
8.     összeg += osztály[tag]['kor']
9.
10. print('Az osztály átlagéletkora', összeg/len(osztály), 'év.')
```

Ránézésre tudjuk, hogy mit csinál a kód! ☺

2. A lányok vannak többen, vagy a fiúk?
   A megoldást nem mutatjuk meg teljes egészében – az előző lecke 9. feladatának kiírást végző része változtatás nélkül használható itt is.

```
6. lányok = 0
7. for tag in osztály:
8.     if osztály[tag]['nem'] == 'l':
9.         lányok += 1
```

kis L betű

A vonósnégyes és a szótár

Elgondolkodunk még azon, hogy miként érdemes megadnunk az előző lecke vonósnégyesének jelenléti ívét. Lehet például egy olyan szótárunk, amelyikben a kulcsok az egyes zenészek, az értékek pedig a jelenléti listák:

```
1. ív = {'egyik hegedűs': [1, 1, 1, 1, 1],
2.       'másik hegedűs': [1, 1, 1, 1, 0],
3.       'brácsás': [1, 1, 0, 0, 0],
4.       'csellós': [0, 1, 1, 1, 1]}
```

Így már valóban tudunk válaszolni arra, hogy *ki* hiányzott a legtöbbet (és nem csak sorszámot adunk), és mi annak a napnak a *neve*, amikor senki nem hiányzott a próbáról. Természetesen ezúttal is lehetőség van az egyes zenészek listáit újabb szótárakká alakítani. Ekképp látszanának az egyes napok nevei.

Sok esetben van tehát értelme szótárakat használnunk, de sok az olyan eset is, amikor a szótárak használata csak bonyolítaná a programunkat. Az okos fejlesztő végiggondolja a lehetőségeit, mielőtt nekikezd egy program megvalósításának.

Kétdimenziós listák és szótárak a gyakorlatban

Híres szakemberek az adatszerkezetekről

Fred Brooks, a világ egyik leghíresebb számítógéptudósa mondta még 1975-ben:

„Mutasd meg a folyamatábráidat, és rejtsd el a táblázataidat, és homályban maradok. Mutasd meg a táblázataidat, és rendszerint nem kellenek majd a folyamatábrák – minden nyilvánvaló lesz.” – Mai szemmel talán érdemes a folyamatábra helyére a „kód” szót, a táblázat helyére az „adatszerkezet” szót tennünk, és látjuk, hogy tényleg így van, mind a mai napig.

Linus Torvalds, az első Linux megalkotója és a Linux kernel karbantartását végző, mára hatalmas fejlesztői csapat feje szerint „A rossz programozók aggódnak a kódjuk miatt. A jó programozók az adatszerkezeteikkel és azok kapcsolataival törődnek.”

Eric S. Raymond szerint „Az okos adatszerkezet és a buta kód sokkal jobban működik, mint fordítva.”

Guido van Rossum, a Python megalkotója pedig arra figyelmeztet: „Könnyű olyan hibákat elkövetni, amik csak később jönnek elő, miután rengeteg kódot megírtunk. Az ember ráébred, hogy másik adatszerkezetet kellett volna használni. Kezdhetjük elölről.”

Feladatok

Rendelkezésünkre áll egy bolthálózat négy boltjának ezer forintban kifejezett bevétele az elmúlt hét napból.

130	29	143	133
156	15	222	132
231	210	98	182
112	11	101	121
96	191	184	148
311	14	201	199
231	302	87	187

Írjunk olyan programot, eljárást, függvényt, ami az alábbi kérdésekre válaszol!

1. Melyik boltnak a legnagyobb a bevétele az elmúlt hét napban?
2. Melyik boltnak a legnagyobb a tegnapi bevétele?
3. Melyik boltban legnagyobb a legjobb és a legrosszabb nap közötti különbség?

```
1. bevételek = [[130,156,231,112,96,311,231],
2.              [29,15,210,11,191,14,302],
3.              [143,222,98,101,184,201,87],
4.              [133,132,182,121,148,199,187]]
5.
6. legnagyobb_különbség = 0
7. legnagyobb_különbség_helye = None
8. for index in range(len(bevételek)):
```