Algoritmizálás és programozási nyelv használata

Az *Algoritmizálás és programozási nyelv használata* című fejezet átfogó ismétléssel kezdődik. Ebben a kötetben már megismerkedünk az elemi típusalgoritmusok strukturált programozásban használatos formájával. A komolyabb, nehezebb programozási ismeretek közül elsőként a fájlkezelés kerül sorra, mert remekül használható az összetett típusalgoritmusok megismerésekor. Az algoritmusok közül erősebb hangsúlyt kapnak a középiskolai szintű feladatokban gyakrabban előfordulók.

Az emelt szintű ismereteket taglaló témaköröket követően a fejezet tárgyalási módja változik. Az objektumorientált programozásról és a grafikus felület programozásáról szólva a sok rövid példa helyett egy-két feladat részletesebben ismertetett megoldására hagyatkozunk, melyekben ismerkedni kezdünk az olyan problémamegoldási attitűddel és módszerkészlettel, amilyet a nagyobb lélegzetű feladatok megoldása gyakran megkíván.

Vittük valamire – Szekvenciák, elágazások és a feltételes ciklus

Kilencedik és tizedik évfolyamon jócskán belekóstoltunk a programozásba, és már egészen klassz dolgokra tudjuk rávenni a számítógépet – ebben és a soron következő három leckében először ezeket az ismereteket ismételjük át.

A bennünket körülvevő rengeteg számítógép közül a digitáliskultúra-órákon minket épp azok érdekelnek leginkább, amelyekről látszik, hogy számítógépek: a laptopok és az asztali számítógépek. Mindazonáltal tudjuk, hogy éppúgy programok működtetik a háztartási gépeinkben, járműveinkben, egyéb eszközeinkben lévő számítógépeket is, és az sem titok, hogy mára csak a legegyszerűbb gépeinkben nincs számítógép.

Az említett programokat sokféle nyelven írhatjuk – itt a könyvben történetesen a Pythonban, ha úgy tetszik, pythonul fogalmazzuk meg a programjainkat. Legyen azonban szó bármelyik programozási nyelvről, ha elég messziről nézzük őket, igencsak hasonlóak. Az előző mondatnak csak látszólag mond ellent az a megjegyzésünk, hogy az egyes nyelvek bizonyos feladatok elvégzésére alkalmasabbak lehetnek a többinél. Az újabb és újabb programozási nyelvek, illetve környezetek sokszor a visszatérő problémák kész megoldását nyújtják függvények, eljárások formájában.

A programok mindig utasításokból állnak. A legtöbb nyelv használatakor – ha másképp nem kérjük – a számítógép ezeket az utasításokat szépen sorra veszi, és egymás után – programozóul úgy mondjuk, hogy **szekvenciában** – végrehajtja őket.

A sorszámokat nem írjuk be, csak a könyvben számozzuk, mert így könnyebb elmondani, hogy melyikről beszélünk.

```
1. #!/usr/bin/env python3
2.
3. print('Idén is lesz programozás.')
4. print('Hát nem remek? ;)')
```

Ez a sor csak Linuxon és macOS-en kell, Windowson nem fontos. Mostantól a könyvben nem írjuk ki.

Már a legegyszerűbb programok is képesek lehetnek a felhasználóval való kommunikációra. A felhasználó válaszait (is) változókban tároljuk. A **változók** értéke a változó nevének említésével kiolvasható.

Gyakori, hogy egy változó értékétől függően mást és mást csinál a programunk. Ilyenkor **elágazás** van a programban, aminek csak az egyik ágát szeretnénk végrehajtani.

Változót hozunk létre, és elhelyezzük benne a felhasználó válaszát.

```
1. print('Idén is lesz programozás.')
2. vélemény = input('Örülsz? (i/n) ')
3. if vélemény == 'i':
4.     print('Hát én is!')
5.     print('Jaj, de jó!')
6. elif vélemény == 'n':
7.     print('Hüpp.')
8. else:
9.     print('Nem értem. Pedig igazán próbálkoztam.')
10. print('Pápá.')
```

Ez a két sor akkor fut le, ha a felhasználó „i”-t válaszolt.

Ez akkor, ha „n”-t.

Ez minden egyéb esetben.

Ez pedig mindig lefut, mert nincs behúzva, azaz az elágazás után következik.

1. példa: Cica vagy kutya?

Írjuk meg azt a programot `cicakutya` néven, amelyik megkérdi a felhasználót, hogy a program cica vagy kutya legyen-e! Ha a felhasználó cicát szeretne, akkor a program kérjen tejet, és írja ki, hogy „Nyaú!”. Kutya esetén persze csontra lesz szükség, a megnyilvánulás pedig „Vaú!”.

```
1. állat = input('Cica vagy kutya legyek? (c/k) ')
2. if állat == 'c':
3.     print('Tejet, ha lehet.')
4.     print('Nyaú!', end='')
5. elif állat == 'k':
6.     print('Egyet mondok, adjál csontot!')
7.     print('Vaú!', end='')
```

Ha valamilyen ismétlődő feladatot szeretnénk végeztetni a számítógéppel, **ciklust** szervezünk. Az első ciklusunk az **elöltesztelő feltételes ciklus** vagy `while`-ciklus. Ebbe a ciklusba akkor lépünk be, ha a belépés feltétele igaz, és addig maradunk benne, amíg ez a feltétel teljesül.

Ha a felhasználó „i”-t válaszolt, belépünk a ciklusba, és cincogunk.

```
1. legyen_cincogás = input('Üdv, én egy egér vagyok. Cincogjak neked? (i/n) ')
2. while legyen_cincogás == 'i':
3.     print('Cin-cin!')
4.     legyen_cincogás = input('Még? (i/n) ')
5. print('Szia.')
```

Ha itt „n”-t (pontosabban: nem „i”-t) válaszolunk, akkor a következő ismétlődésnél a 2. sorban nem teljesül a feltétel, és nem maradunk a ciklusban.

2. példa: A harmincéves háború időtartama

Írjunk programot, amely addig kérdezgeti, hogy hány évig tartott a harmincéves háború, amíg a felhasználó ki nem találja! Ha a programozási nyelvünk miatt szükség van rá, ne felejtsük megfelelő típusúra alakítani a választ! Segítségül itt az algoritmus mondatszerű leírása:

```
program haboru30:
    válasz = speciális érték, ami biztosan nem 30
    ciklus amíg válasz <> 30
        be: válasz
    ciklus vége
    ki: dicséret
program vége
```

Amikor már működik a programunk, itt az ideje továbbfejleszteni. Ha a felhasználónk tippje hibás, írjuk ki, hogy kisebb vagy nagyobb-e a megfelelő érték! Ha pedig harmadjára sem találta el, megsúghatjuk neki, hogy mettől meddig tartott a szóban forgó háború.

```
1. válasz = None
2. próbálkozások = 0
3.
4. while válasz != 30:
5.     if próbálkozások >= 3:
6.         print('Súgok: 1618-tól 1648-ig tartott.')
7.     válasz = input('Hány évig tartott a harmincéves háború? ')
8.     válasz = int(válasz)
9.     próbálkozások += 1
10.    if válasz > 30:
11.        print('Rövidebb volt.')
12.    elif válasz < 30:
13.        print('Hosszabb volt.')
14.
15. print('Ügyes! Nem gondoltam volna...')
```

Létrehozzuk a változót (mert a negyedik sorban már kelleni fog), de nem adunk neki értéket.

Vittük valamire – Bejárható objektumok és a bejárós ciklus, eljárások és függvények

Az előző leckében a feltételes ciklussal is foglalkoztunk. A másik ciklusunk – merthogy Pythonban kétféle van – a **bejárós** vagy `for`-ciklus. A feltételes ciklusnak adnunk kell egy bejárható objektumot, amelynek az elemein végiglépdelhet, azaz amit bejárhat. Eddig négyféle bejárható objektummal ismerkedtünk meg. Lássuk őket sorban:

A **range típusú objektumok** egy számsort tartalmaznak, és a `range()` függvénnyel tudunk ilyet előállítani. A bejárós ciklus **ciklusváltozója** range-objektum bejárásakor az egyes számokat kapja értékül.

3. példa: Szökőévek I. Ferenc uralkodásától napjainkig

Írjunk programot, amely megadja I. Ferenc József trónra lépésétől az idei évig tartó időszak szökőéveit!

```
1. idei_év = 2022
2. for év in range(1848, idei_év+1):
3.     if év % 4 == 0 and \
4.         (év % 100 != 0 or év % 400 == 0):
5.         print(év, '.', sep='')
```

Listák és karakterláncok

A **listák** összetartozó adatokat tartalmaznak. Amikor listát járunk be `for`-ciklussal, akkor a ciklusváltozó az egyes listaelemeket kapja értékül.

A **karakterláncok,** azaz stringek a listákhoz hasonlóan viselkednek. Egy string bejárásakor a ciklusváltozó az egymás utáni karakterek értékét veszi fel. Karakterláncok és listák között fontos különbség, hogy csak a listák módosíthatók. Új listaelemek hozzáadásával bővíthetők, a meglévő elemeik cserélhetők vagy törölhetők. Mindez a stringek esetében nem lehetséges.

Az alábbi kód a lista és a karakterlánc adattípus pár tulajdonságát mutatja be.

```
1. szöveg = input('Írj ide egy szót kisbetűkkel! ')
2. szöveg = szöveg.upper()
3.
4. lista = []
5.
6. for betű in szöveg:
7.     print('A(z)', betű, 'kerül a listába.', end=' ')
8.     lista.append(betű)
9.     print('A lista értéke most: ', lista)
```

A stringeknek van néhány remek tagfüggvényük. A „python string methods" kifejezésre keresve találunk róluk információt.

A listáknak is van néhány remek tagfüggvényük. Hasonló kereséssel ezeket is megleljük az interneten.

Feladat

Keressük meg az interneten, hogy mi minden közös a listák és a karakterláncok kezelésében, és nevezzünk meg néhány eltérést is!

4. példa: A három kismalac háza

Listákból használtunk kétdimenziósakat is – ezek olyasmik, mint a táblázatok. Lássunk egy olyat (`malacok.py`), amely a három kismalac nevét és házának anyagát tárolja.

```
1. malacok = [['Töfi', 'szalma'],
2.            ['Röfi', 'fa'],
3.            ['Döfi', 'kő']]
```

A „nagy” lista három kisebb listát tartalmaz.

Amikor egy ilyen listát bejárunk, a ciklusváltozóba az épp aktuális kis lista kerül.
Írjuk ki egy-egy sorba az egyes malacok nevét és a házuk anyagát! Amelyik malacnak „kő”-ből készül a háza, amellé írjuk oda azt is, hogy a farkas nem tudja bántani! Teszteljük a programot úgy, hogy a listába újabb kőházas malacokat veszünk fel!

```
4. for malac in malacok:
5.     print(malac[0], ' házának anyaga: ' , malac[1], '.', sep='', end='')
6.     if malac[1] == 'kő':
7.         print(' A farkas nem tudja bántani ', malac[0], 't.', sep='')
8.     else:
9.         print()
```

Bár a programunk remekül működik, egy idő után nehéz lehet követni, hogy mit is jelent a `malac[0]` és a hasonló jelölések. A **szótárak** pont ezen a problémán segítenek. A szótárakban az egyes objektumoknak különféle tulajdonságait tárolhatjuk:

```
1. malacok = [{'név': 'Töfi', 'ház_anyag': 'szalma'},
2.            {'név': 'Röfi', 'ház_anyag': 'fa'},
3.            {'név': 'Döfi', 'ház_anyag': 'kő'}]
```

Egy-egy malacot egy-egy szótárban tárolunk.

A három szótár az előző „nagy” listában van.

Módosítsuk úgy a kétdimenziós listát tartalmazó malacos programot, hogy ezúttal szótárakkal dolgozzon!

```
4. for malac in malacok:
5.     print(malac['név'], ' házának anyaga: ' , malac['ház_anyag'], '.', sep='', end='')
6.     if malac['ház_anyag'] == 'kő':
7.         print(' A farkas nem tudja bántani ', malac['név'], 't.', sep='')
8.     else:
9.         print()
```

A szótárak szintén bejárható objektumok. Bejárhatók úgy, hogy a kulcsaikat kapjuk meg (az előző példában „név” és „ház_anyaga”), de bejárhatók érték szerint vagy épp mindkettőt elkérve. Lássunk egy példát:

A szótárakban a kulcsok egyediek – nem lehet két ,,név'' vagy két ,,magasság'' egyetlen szótárban.

```
1. ember = {'név': 'Debóra', 'magasság': 167, 'IQ': 131}
2.
3. for kulcs in ember.keys(): # vagy csak: in ember
4.     print(kulcs, 'értéke:', ember[kulcs])
5. print()
6.
7. for érték in ember.values():
8.     print(érték)
9. print()
10.
11. for kulcs, érték in ember.items():
12.     print(kulcs, 'értéke:', érték)
```

A kulcs szerinti bejáráskor a ciklusváltozóba az épp aktuális elem kulcsa kerül. A kulcs ismeretében megkaphatjuk az értéket is.

Így a kulcsot és az értéket is megkapjuk.

5. példa: Kedvenceink

Írjunk egy programot `kedvencek` néven, amely egy-egy szótárban tárolja három kedvenc vloggerünket, előadónkat, tanárunkat vagy sakknagymesterünket! Egy szótárnak három kulcsa legyen:

- a tárolt ember neve,
- legjelentősebb teljesítménye a szakmájában és
- az említett teljesítmény évszáma.

A három szótárat helyezzük el egy listában, majd járjuk be a listát, és írjuk ki, amit az egyes emberekről tudunk!

```
1. nagymesterek = [{'név': 'Polgár Judit',
2.                  'teljesítmény': 'Garri Kaszparov legyőzése',
3.                  'év': 2002},
4.                 {'név': 'Kempelen törökje',
5.                  'teljesítmény': 'Napóleon legyőzése (sakkban)',
6.                  'év': 1809},
7.                 {'név': 'Beth Harmon',
8.                  'teljesítmény': 'TV-sorozattá lett az "élete"',
9.                  'év': 2020}]
10.
11. for nagymester in nagymesterek:
12.     print(nagymester['név'], ': ',
13.           nagymester['teljesítmény'], ' (',
14.           nagymester['év'], ')', sep='')
```

A három szótár

Listában vannak.

Így nem kezd új sort a print().

Ha a programunk szépen működik, akkor úgy bővítjük, hogy a felhasználónak legyen lehetősége új embereket felvenni a listába. A listát az új emberek felvétele előtt és után is ki akarjuk írni. Ha egy programban több helyen csinálnánk ugyanazt, akkor eljárást vagy függvényt készítenénk a feladat elvégzésére. A kettő között az a különbség, hogy a függvénynek van visszatérési értéke, az eljárásnak nincs. Sok nyelv, így a Python sem különbözteti meg az eljárást a függvénytől: az eljárást egyszerűen olyan függvénynek tekinti, amelynek nincs visszatérési értéke.

Alapvetően nem jó, ha az eljárások és a függvények a főprogram változóihoz nyúlnak. Okosan tesszük, ha paraméterként átadjuk számukra azokat a változókat, amikkel dolgunk van.

Alakítsuk át a főprogram 11–14. sorát! Most, hogy eljárás lett belőlük, érdemes lehet a program elejére helyezni őket.

```
1. def nagymestereket_listáz(nmk):
2.     for nm in nmk:
3.         print(nm['név'], ': ', nm['teljesítmény'], ' (', nm['év'], ')', sep='')
```

Ezzel megszületik a `nagymestereket_listáz` eljárásunk, amelynek a paramétere az „`nmk`” nevű lista. Az eljárás bejárja a listát, és kiírja az elemeit. Elterjedt szokás – bár a Pythonban nem kötelező – az eljárásokat és a függvényeket a program elején elhelyezni. Az eljárások lefuttatása, azaz hívása az eljárás nevével történik. A programunkba szúrjuk be az alábbi sort!

```
11. nagymestereket_listáz(nagymesterek):
```

Híváskor a „nagymesterek” listát adjuk át – de az eljárás már „nmk” néven fog a megkapott listával dolgozni.

Ha most is minden szépen működik, akkor átgondoljuk, hogy egy új ember megadásához mit kell tenni:

- három adatot be kell kérni,
- az évszámot számmá kell alakítani,
- az adatokból egy szótárat készíteni és
- a szótárat a lista végére fűzni.

Az első három feladat megoldását egy függvénybe szervezzük. A függvényt csak egyszer fogjuk lefuttatni, de olvashatóbb lesz így a kód, jobban elkülönül, hogy a program melyik része mit csinál – ez a másik eset, amikor eljárást, függvényt használunk. A függvényünknek nincs paramétere, de van visszatérési értéke – hiszen ettől függvény –, mégpedig az új ember adatait tartalmazó szótár.

```
1. def adatokat_bekér():
2.     név = input('Mi a neve? ')
3.     teljesítmény = input('Mi a legjelentősebb teljesítménye? ')
4.     év = int(input('Melyik évben érte ezt el? '))
5.     szótár = {'név': név, 'teljesítmény': teljesítmény, 'év': év}
6.     return szótár
```

A függvényt a programunk végén hívjuk. A főprogram (amelyet megelőz a függvény és az eljárás definíciója, valamint a lista eredeti változata) most ilyen:


```
22. nagymestereket_listáz(nagymesterek)
23.
24. új_ember = adatokat_bekér()
25. nagymesterek.append(új_ember)
26.
27. print('A nagymesterek listája ilyen lett:')
28. nagymestereket_listáz(nagymesterek)
```

Már csak annyi dolgunk maradt, hogy egy ciklussal újra meg újra megkérdezzük, hogy akar-e még a felhasználónk újabb embert megadni. Íme a teljes kód:

```
 1. def adatokat_bekér():
 2.     név = input('Mi a neve? ')
 3.     teljesítmény = input('Mi a legjelentősebb teljesítménye? ')
 4.     év = int(input('Melyik évben érte ezt el? '))
 5.     szótár = {'név': név, 'teljesítmény': teljesítmény, 'év': év}
 6.     return szótár
 7.
 8. def nagymestereket_listáz(nmk):
 9.     for nm in nmk:
10.         print(nm['név'], ': ', nm['teljesítmény'], ' (', nm['év'], ')', sep='')
11.
12. nagymesterek = [{'név': 'Polgár Judit',
13.                  'teljesítmény': 'Garri Kaszparov legyőzése',
14.                  'év': 2002},
15.                 {'név': 'Kempelen törökje',
16.                  'teljesítmény': 'Napóleon legyőzése (sakkban)',
17.                  'év': 1809},
18.                 {'név': 'Beth Harmon',
19.                  'teljesítmény': 'TV-sorozattá lett az "élete"',
20.                  'év': 2020}]
21.
22. nagymestereket_listáz(nagymesterek)
23.
24. bővítjük = None
25. while bővítjük != 'n':
26.     bővítjük = input('Akarsz új nagymestert megadni? (i/n) ')
27.     if bővítjük == 'i':
28.         nagymesterek.append(adatokat_bekér())
29.
30. print('A nagymesterek listája ilyen lett:')
31. nagymestereket_listáz(nagymesterek)
```

Vittük valamire – Típusalgoritmusok

A tizedik évfolyamos könyvünkben láttuk, hogy a típusalgoritmusokat – más néven programozási tételeket – azért érdemes ismerni, mert gyakran felmerülő problémákra adnak kipróbált megoldást. Az eddig megismert – úgynevezett „egyszerű” – típusalgoritmusok közös jellemzője, hogy egy bejárható objektumot vizsgálnak, és egy egyszerű értékre hivatkozva térnek vissza. Az egyszerű érték lehet például egy szám, a sorozat egy eleme vagy logikai érték: igaz vagy hamis.

Vannak olyan programozási nyelvek – ilyen a Python is –, ahol a típusalgoritmusoknak van rövid formájuk, függvényük is. A rövid formákat nagyon szeretjük (gyorsabban lehet beírni őket, könnyen olvashatók), de ebben a mostani könyvünkben is hangsúlyozzuk: sokszor olyan, összetettebb eset van, amikor csak a hosszabb, általánosabb forma használható. Fontos tehát, hogy a megoldás elvét is megismerjük. Így mindig pontosan az épp felmerülő problémának megfelelően tudjuk alkalmazni, kódolni a típusalgoritmusokat.

Sorozatszámítás

A sorozatszámítás algoritmusa arra jó, hogy egy bejárható objektum elemeit adjuk össze, szorozzuk össze, vagy írjuk egymás mellé. Azaz

ebből	ilyen művelettel	ilyet készít:
`[1,2,5]`	összeadás	`8`
`['ma', 'j', 'om']`	összefűzés	`'majom'`
`[2,3,4]`	átlagolás	`3`
`[1,2,3,4]`	összeszorzás	`24`

6. példa: Adjuk meg az év hosszát az egyes hónapok napjainak száma alapján!

```
1. hónapok_napjai = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
2.
3. #mindig használható megoldás
4. napok_száma = 0
5. for hónap in hónapok_napjai:
6.     napok_száma += hónap
7. print(napok_száma)
8.
9. #egyszerű megoldás:
10. napok_száma = sum(hónapok_napjai)
11. print(napok_száma)
```

Melyik fent említett művelethez tudunk még rövid formát adni?

Eldöntés

Az eldöntés algoritmusa arra a kérdésre ad választ, hogy van-e adott tulajdonságú elem a bejárható objektumunkban.

7. példa: Van-e 28 napos hónap az évben?

Az első megoldásunk feltételes ciklussal dolgozik, és így megfelel a *strukturált programozás* követelményeinek.

```
1. hónapok_napjai = [31,28,31,30,31,30,31,31,30,31,30,31]
2.
3. van_28 = False
4. index = 0
5. hossz = len(hónapok_napjai)
6. while index < hossz and not van_28:
7.     if hónapok_napjai[index] == 28:
8.         van_28 = True
9.     index += 1
10. if van_28:
11.     print('Van 28 napos hónap.')
12. else:
13.     print('Nem találtunk 28 napos hónapot.')
```

Itt tároljuk, hogy találtunk-e a keresett értékből.

A ciklus addig fut, amíg nem találunk olyan értéket, amilyet keresünk.

Ha találtunk, feljegyezzük.

Elágazás helyett használhatunk logikai kifejezést az értékadásra. A fenti kód hetedik sora ilyenkor a `van_28 = (hónapok_napjai[index] == 28)` formát ölti, cserébe nem kell a nyolcadik. A `hónapok_napjai[index] == 28` logikai kifejezés kiértékeléskor igaz vagy hamis lesz, és ezt az értéket kapja értékül a `van_28` változó.

A második megoldásunk bejárós ciklust használ. Most még kevéssé hatékony, mert bejárja az összes értéket. (A többi megoldást csak a harmadik sortól közöljük, lévén az első kettő változatlan.)

```
3. van_28 = False
4. for hónap in hónapok_napjai:
5.     if hónap == 28:
6.         van_28 = True
7. if van_28:
8.     print('Van 28 napos hónap.')
9. else:
10.     print('Nem találtunk 28 napos hónapot.')
```

A ciklus bejárja az összes értéket.

A továbbfejleszett változat már hatékony: ha talál a keresett értékből, akkor megszakítja a ciklust.

```
3. van_28 = False
4. for hónap in hónapok_napjai:
5.     if hónap == 28:
6.         van_28 = True
7.         break
8. if van_28:
9.     print('Van 28 napos hónap.')
10. else:
11.     print('Nem találtunk 28 napos hónapot.')
```

A ciklus bejárná az összes értéket.

Ha megvan, amit keresünk, félbehagyjuk a ciklust.

A Python nyelv úgy segíti elő a `break` utasítás használatát, azaz a ciklusokból való idő előtti kilépést, hogy a ciklusoknak létezik `else`-záradékuk – ez meglehetősen ritka dolog a programozási nyelvek körében. Az `else`-záradék csak akkor fut le, ha a ciklus végigfutott, azaz nem léptünk ki belőle `break`-kel. Ezt használjuk az utolsó megoldásunkban:

```
3. for hónap in hónapok_napjai:
4.     if hónap == 28:
5.         print('Van 28 napos hónap.')
6.         break
7. else:
8.     print('Nem találtunk 28 napos hónapot.')
```

Ez az else a for-hoz tartozik! Az else-ág kódja csak akkor fut le, ha nem volt break.

A fenti megoldások ugyanarra az eredményre vezetnek. Vannak, akik a bejárós ciklust használó megoldást könnyebben olvassák és értik, az elöltesztelő ciklust használó kód pedig bizonyítottan helyes megoldása a problémának.

Természetesen az eldöntésre is van egyszerű megoldás:

```
3. if 28 in hónapok_napjai:
4.     print('Van 28 napos hónap.')
```

Kiválasztás

A kiválasztás algoritmusa akkor használható, ha tudjuk, hogy van adott tulajdonságú elem a bejárható objektumunkban, és kíváncsiak vagyunk, hogy hányadik sorszámú ez az elem.

8. példa: Hányadik hónap a 28 napos?

Elsőként – a strukturált programozás követelményének eleget téve – elöltesztelő feltételes ciklust használunk.

A ciklus addig fut, amíg meg nem találjuk, amit keresünk.

```
1. hónapok_napjai = [31,28,31,30,31,30,31,31,30,31,30,31]
2.
3. index = 0
4. while hónapok_napjai[index] != 28:
5.     index += 1
6. print('A(z) ', index+1, '. hónap 28 napos.', sep='')
```

A kiválasztás is megvalósítható bejárós (számlálós) ciklussal. A kiírás kerülhet a ciklusba is, és utána is – írjuk oda, ahol az adott programban jobb helye van!

```
3. for index in range(len(hónapok_napjai)):
4.     if hónapok_napjai[index] == 28:
5.         break
6. print('A(z) ', index + 1, '. hónap 28 napos.', sep='')
```

Pythonban tudunk olyan bejárós ciklust írni, amelyik bejárja a kapott objektumot, és a bejárt értékeket számozza is. Ezt tekintjük pythonosabb megoldásnak:

két ciklusváltozó

enumerate = számozd be!

```
3. for index, nap in enumerate(hónapok_napjai):
4.     if nap == 28:
5.         break
6. print('A(z) ', index + 1, '. hónap 28 napos.', sep='')
```

És most is van egyszerű formánk:

```
3. print('A(z) ', hónapok_napjai.index(28)+1, '. hónap 28 napos.', sep='')
```

Keresés

A keresés algoritmusa az eldöntés és a kiválasztás egybeépítése. Van-e adott tulajdonságú elem, és ha igen, akkor hol?

9. példa: Ha van 28 napos hónap az évben, akkor hányadik hónap ilyen?

A megoldás feltételes ciklussal:

Ha nem találtunk 28 napos hónapot, akkor az index változó a lista utolsó elemén túlra fog mutatni, de ebből nem lesz baj, mert a while feltételben figyelünk rá.

```
1. hónapok_napjai = [31,28,31,30,31,30,31,31,30,31,30,31]
2.
3. index = 0
4. while index < len(hónapok_napjai) and hónapok_napjai[index] != 28:
5.     index += 1
6. if index < len(hónapok_napjai):
7.     print('A(z) ', index+1, '. hónap 28 napos.', sep='')
8. else:
9.     print('Nem találtunk 28 napos hónapot.')
```

Amikor nem találunk 28-at a listában, az index változó túlmutat a lista végén. Most nem mutat túl, azaz találtunk.

Bejárós ciklust használva:

```
3. hányadik_28 = None
4. for index in range(len(hónapok_napjai)):
5.     if hónapok_napjai[index] == 28:
6.         hányadik_28 = index
7.         break
8. if hányadik_28:
9.     print('A(z) ', hányadik_28 + 1, '. hónap 28 napos.', sep='')
10. else:
11.     print('Nem találtunk 28 napos hónapot.')
```

azaz:
if hányadik_28 != None

A következő megoldásban ismét két változóval járjuk be a ciklust, és a `for`-ciklushoz tartozó `else`-záradékot is használjuk:

```
3. for index, nap in enumerate(hónapok_napjai):
4.     if nap == 28:
5.         print('A(z) ', index + 1, '. hónap 28 napos.', sep='')
6.         break
7. else:
8.     print('Nem találtunk 28 napos hónapot.')
```

Az egyszerű megoldás természetesen az előző két típusalgoritmus egyszerű változatának egybeépítését jelenti:

```
3. if 28 in hónapok_napjai:
4.     print('A(z) ', hónapok_napjai.index(28)+1, '. hónap 28 napos.', sep='')
5. else:
6.     print('Nem találtunk 28 napos hónapot.')
```

Megszámolás

A megszámolás algoritmusa megmondja, hogy hány adott tulajdonságú elem van a bejárható objektumban.

10. példa: Hány 30 napos hónap van az évben?

```
1. hónapok_napjai = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
2.
3. #mindig használható megoldás
4. hány_30 = 0
5. for hónap in hónapok_napjai:
6.     if hónap == 30:
7.         hány_30 += 1
8. print(hány_30, '30 napos hónap van.')
9.
10. #egyszerű megoldás:
11. hány_30 = hónapok_napjai.count(30)
12. print(hány_30, '30 napos hónap van.')
```

Írhatunk most breaket?

Szélsőérték-kiválasztás: maximum- és a minimumkiválasztás

A maximum- és a minimumkiválasztás a legkisebb és a legnagyobb elemet keresi meg.

11. példa: Hány napos az év legrövidebb hónapja?

```
1. hónapok_napjai = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
2.
3. #mindig használható megoldás
4. legrövidebb = 32
5. for hónap in hónapok_napjai:
6.     if hónap < legrövidebb:
7.         legrövidebb = hónap
8. print('A legrövidebb hónap', legrövidebb, 'napos.')
9.
10. #egyszerű megoldás:
11. legrövidebb = min(hónapok_napjai)
12. print('A legrövidebb hónap', legrövidebb, 'napos.')
```

Ide olyan értéket adunk meg, ami biztos nem lehet a minimum. 32 napos hónap remélhetőleg nincs a listánkban.

Az előző példákban mindig használható volt a rövidebb, függvényes megoldási forma. Általános szabály, hogy a függvényes megoldás minden olyan esetben alkalmazható, amikor

pontosan azt az értéket akarjuk összeadni, megtalálni, megszámolni, ami a bejárható objektumban van. Ha nem az ott szereplő elemmel, hanem

- a belőle kiszámítható, meghatározható értékkel,
- vagy az elemek közül csak azokkal, amelyek megfelelnek valamilyen feltételnek,

tehát *nem mindegyikkel* kell dolgoznunk, az egyszerűbb, függvényes formák mit sem érnek.

Feladatok

A következő kérdések egy tíznapos periódus hajnali hőmérsékleteit tartalmazó listára vonatkoznak.

Keressünk három olyan kérdést, amely megoldható valamelyik típusalgoritmus egyszerűbb formájával, és három olyat, amely nem! Ez utóbbiak megoldását mindenképp kódoljuk is!

```
hőmérsékletek = [7, 5, -2, 0, -4, 3, 3, 3, 4, 4]
```

1. Hány fok volt a legmelegebb hajnalon?
2. Volt-e olyan hajnal, amikor fagyott? (A nullafokos víz vagy megfagyott már, vagy még éppen nem. Menjünk biztosra, nézzük a nulla foknál hidegebb hajnalokat!)
3. Volt-e olyan hajnal, amikor három fokot mértek?
4. Hány nullafokos hajnal volt?
5. Mikor volt először nullafokos hajnal?
6. Hányadik hajnalon volt fagy először?
7. Hányadik hajnalon volt utoljára fagy?
8. Hányszor fagyott hajnalban?
9. Volt-e olyan, hogy egymás utáni hajnalokon ugyanazt a hőmérsékletet mérték?
10. Hányadik hajnalon volt először melegebb, mint előző nap?

Az egyes kérdéseket megoldó programok megtalálhatók a tankönyv webhelyéről letöltött fájlok között.

Vittük valamire – Típusalgoritmusok kétdimenziós listákkal és szótárakkal

Amikor többdimenziós listákat vagy szótárakat vizsgálunk típusalgoritmusainkkal, csak egy-egy részfeladatban tudjuk használni a típusalgoritmusokat megoldó függvényeket. Azért van ez így, mert – ahogy már tizedik évfolyamon is láttuk – maguk az értékek nem állnak rendelkezésre azonnal feldolgozható formában.

Ennek a leckének az első részében az alábbi kétdimenziós listával dolgozunk:

```
1. hiányzók = [[3, 4, 0, 0, 1],
2.             [2, 3, 0, 0, 0],
3.             [4, 3, 2, 3, 4],
4.             [0, 0, 1, 0, 0]]
```

A nagy listában négy kisebb lista van.

12. példa: Hiányzók

A `hiányzók` lista egy négyhetes időszak egyes napjain mutatja a 11. zs osztály tanulóinak hiányzásait: például a második hét keddjén három fő hiányzott. A lista megadását a kódban az olvashatóság kedvéért egy üres sor követné, úgyhogy a következő kódok mind a 6. sortól kezdődnek majd.

1. Hány hiányzás volt összesen?

```
6. összes = 0
7. for hét in hiányzók:
8.     for nap in hét:
9.         összes += nap
```

A sorozatszámítást egymásba ágyazott ciklusok belsejében végezzük.

Ha szemfülesek vagyunk, észrevehetjük, hogy a belső ciklust kiválthatjuk az algoritmust megoldó függvénnyel.

```
6. összes = 0
7. for hét in hiányzók:
8.     összes += sum(hét)
```

2. Volt-e olyan hét, amikor nem volt hiányzó?

Ezúttal is hetenként járjuk be a listánkat. Mielőtt a bejárást megkezdenénk, ahogy az eldöntéskor szokás, kezdeti értékkel megadunk egy változót. A kezdeti érték megadásakor feltételezzük, hogy még nem volt hiányzásmentes hét, így az érték most hamis lesz, és ha találunk hiányzásmentes hetet, majd igazra állítjuk.

Ha már az imént szemfülesek voltunk, legyünk most is azok! Vegyük észre, hogy egy hét akkor hiányzásmentes, ha a heti hiányzások összege 0.

Megoldás a strukturált programozásnak megfelelő feltételes ciklussal:

```
index = 0
hossz = len(hiányzók)
while index < hossz and sum(hiányzók[index]) != 0:
    index += 1
if index < hossz:
    print('Volt hiányzásmentes hét.')
else:
    print('Nem volt hiányzásmentes hét.')
```

Sorozatszámítás az eldöntés belsejében!

Megoldás bejárós ciklussal:

```
for hét in hiányzók:
    if sum(hét) == 0:
        print('Volt hiányzásmentes hét.')
        break
else:
    print('Nem volt hiányzásmentes hét.')
```

3. Volt-e olyan hét, amikor ötnél kevesebb hiányzás volt?
Ugye látjuk, hogy egy relációs jel módosítása a lényegi változtatás az előző feladat kódjához képest?

4. Melyik héten volt a legtöbb hiányzás?
A feladat egyszerű lenne, ha rendelkezésre állnának a *heti* hiányzások számai. Nem állnak, de ki tiltja meg, hogy egy újabb listába kigyűjtsük magunknak őket?

```
heti_hiányzások = []

for hét in hiányzók:
    heti_hiányzások.append(sum(hét))

legtöbb = max(heti_hiányzások)
print('A(z) ', heti_hiányzások.index(legtöbb)+1, '. héten volt a legtöbb hiányzás.', sep='')
```

üres lista

Kigyűjtjük a heti hiányzásokat (összegzés).

maximumkiválasztás

keresés

A lecke elején azt mondtuk, ennyire összetett adatszerkezetekben kevés esetben használhatjuk majd a típusalgoritmusok egyszerű formáit, viszont a fenti hét sorban háromszor is használtunk egyet-egyet. Ez igaz, de mindhárom esetben egy egydimenziós listával dolgoztunk.

5. Hányadik héten volt egyetlen hiányzás?
Az egyik lehetséges megoldás az előző feladat kódjának enyhe módosítása. Ezúttal is kigyűjtjük a heti hiányzásokat, majd egyszerűen megkeressük, hogy hányadik indexű elem az 1. Így lényegében egyszerűbb a feladat, mint az előző – nincs benne maximumkiválasztás.

6. A hétfői vagy a keddi napokon hiányoztak-e többen a négy hét alatt?
Ez a feladat lényegében két, párhuzamosan is elvégezhető sorozatszámítás.

```
 6. hétfői = 0
 7. keddi = 0
 8.
 9. for hét in hiányzók:
10.     hétfői += hét[0]
11.     keddi += hét[1]
12.
13. if hétfői > keddi:
14.     print('Hétfői napokon hiányoznak többen.')
15. elif hétfői < keddi:
16.     print('Keddi napokon hiányoznak többen.')
17. else:
18.     print('Ugyanannyi a hiányzás a hétfői és keddi napokon.')
```

egyik sorozatszámítás

másik sorozatszámítás

13. példa: Kutyaoltás

A szótárakban tárolt objektumok esetében hasonló problémákkal kerülünk szembe, mint az előbb. Ha nagyon egyszerűen nézzük, kétféle esetben használunk szótárat. Az első eset az, amikor **egyetlen szótárat használunk sok azonos jellemző tárolására.** Ilyen eset például, hogy kinek hányas a dolgozata, melyik napon hányan néztek meg egy videót, vagy melyik napon hány kutyát oltott be az állatorvos. Ez utóbbi egy lehetséges szótára:

```
1. oltások = {'hétfő': 8, 'kedd': 14, 'szerda': 2, 'csütörtök': 3, 'péntek':13}
```

A szótár megadását a kódban az olvashatóság okán üres sor követné, így a következő kódok mind a harmadik sortól kezdődnek majd.

7. Hány kutyát oltott be a héten az állatorvos?

```
3. összes = 0
4. for szám in oltások.values():
5.     összes += szám
6. print('A héten', összes, 'kutya kapott oltást.')
```

A szótár értékeit járjuk be.

8. Volt-e olyan nap, amikor legalább tíz kutyát oltott be a doki?

```
3. volt = False
4. for szám in oltások.values():
5.     if szám >= 10:
6.         volt = True
7. if volt:
8.     print('Volt legalább tízkutyás nap.')
```

9. Melyik napon oltotta be a legkevesebb állatot az állatorvos?

```
3. legkevesebb_nap = None
4. legkevesebb_szám = 100000
5. for nap, szám in oltások.items():
6.     if szám < legkevesebb_szám:
7.         legkevesebb_nap = nap
8.         legkevesebb_szám = szám
```

A kulcsokat is bejárjuk, mert azokat kell kiírni.

```
8. print('A legkevesebb kutya ezen a napon kapott oltást:', legkevesebb_nap)
```

Vegyük észre, hogy a szótárak ilyen használata lényegében arra jó, hogy nevet kapjanak az egyes adatok. Használatuk egyéb tekintetben hasonló a listához.

A szótárak használatának másik gyakori esete, amikor nem azonos tulajdonságokat tárolunk. Ilyenkor **egy szótárba egy megfigyelt objektum kerül:** egy fa, egy zeneszám, egy csillag, egy ember. A szótár kulcsai a megfigyelt tulajdonságok: a fa magassága és faja, a csillag színe, távolsága, nagysága, az ember szempilláinak száma és az, hogy bal- vagy jobbkezes. Általában több objektumot tárolunk, és így lényegében egy szótárakból álló listával dolgozunk.

14. példa: Hazánk legmagasabb hegycsúcsai

A következő feladatok során egy olyan listával foglalkozunk, melynek elemei szótár típusúak. A szótárban található objektumok mindegyike országunk legmagasabb hegycsúcsait reprezentálja.

```
1. csúcsok = [{'név': 'Kékes', 'hegység': 'Mátra', 'magasság': 1014},
2.            {'név': 'Hidas-bérc', 'hegység': 'Mátra', 'magasság': 971},
3.            {'név': 'Galya-tető', 'hegység': 'Mátra', 'magasság': 964},
4.            {'név': 'Szilvási-kő', 'hegység': 'Bükk-vidék', 'magasság': 961},
5.            {'név': 'Péter hegyese', 'hegység': 'Mátra', 'magasság': 960},
6.            {'név': 'Kettős-bérc', 'hegység': 'Bükk-vidék', 'magasság': 958},
7.            {'név': 'Istállós-kő', 'hegység': 'Bükk-vidék', 'magasság': 958}]
```

A lista megadását a kódban az olvashatóság kedvéért megint csak egy üres sor követné, úgyhogy a következő kódok mind a 9. sortól kezdődnek majd.

1. Hány csúcs található a Bükk-vidéken?

```
 9. bükki_csúcsok = 0
10.
11. for csúcs in csúcsok:
12.     if csúcs['hegység'] == 'Bükk-vidék':
13.         bükki_csúcsok += 1
14.
15. print('A Bükk-vidék', bükki_csúcsok, 'csúcsa van a listában.')
```

megszámlálás

2. Mennyi a mátrai csúcsok magasságának átlaga?
Nem tudunk egyszerűen a `len()` függvénnyel darabszámot mondani, lévén a listának nem minden eleme számít. Meg kell számlálnunk azokat, amelyek számítanak, és csak azokat kell összegeznünk az átlaghoz.

```
 9. mátra_magasság_összesen = 0
10. mátra_darabszám = 0
11.
12. for csúcs in csúcsok:
13.     if csúcs['hegység'] == 'Mátra':
14.         mátra_magasság_összesen += csúcs['magasság']
15.         mátra_darabszám += 1
16.
17. mátra_átlag = mátra_magasság_összesen / mátra_darabszám
18. print('A mátrai csúcsok átlagosan', round(mátra_átlag), 'méteresek.')
```

sorozatszámítás

megszámlálás

3. Melyik a Bükk-vidék legmagasabb csúcsa?
Az itt közölt megoldásban külön változóban tároljuk az eddigi legmagasabb csúcs magasságát és nevét.

```
 9. bükk_magasság = 0
10. bükk_csúcs = None
11.
12. for csúcs in csúcsok:
13.     if csúcs['hegység'] == 'Bükk-vidék':
14.         if csúcs['magasság'] > bükk_magasság:
15.             bükk_magasság = csúcs['magasság']
16.             bükk_csúcs = csúcs['név']
17.
18. print('A Bükk-vidék legmagasabb csúcsa:', bükk_csúcs)
```

maximumkiválasztás

Másik megoldás lehet, hogy egyetlen változót, a legmagasabb bükki csúcs szótárát tároljuk. Kódoljuk ezt a megoldást is!

Bővítsük úgy a programunkat, hogy kérdezze meg a felhasználótól, hogy melyik hegységet vizsgálja! Ha ez már megy, akkor írja ki a programunk a kérdés elé a lehetséges választási lehetőségeket!

4. Van-e ezer méternél magasabb csúcs a listában?

```
 9. van = False
10.
11. for csúcs in csúcsok:
12.     if csúcs['magasság'] > 1000:
13.         van = True
14.         break
15.
16. if van:
17.     print('Van 1000 méternél magasabb csúcs.')
```

keresés

Ha *pontosan* értjük a kódot, és látjuk, hogy a kiértékelés során a `csúcs['magasság'] > 1000` helyére kerül a `True` vagy `False`, és a 16. sor ezt követően már az `if True` vagy `if False` formát ölti, akkor látjuk azt is, hogy a 11–14. sor átfogalmazható úgy, hogy az értékadáskor egy logikai kifejezést írunk:

```
11. for csúcs in csúcsok:
12.     van = csúcs['magasság'] > 1000
13.     if van:
14.         break
```

Akár az első, akár a második megfogalmazást választjuk, érdemes azt is észrevenni, hogy a kiírás esetleg elhelyezhető a ciklusban:

```
11. for csúcs in csúcsok:
12.     van = csúcs['magasság'] > 1000
13.     if van:
14.         print('Van 1000 méternél magasabb csúcs.')
15.         break
```

Bővítsük a programunkat: kérdezze meg a felhasználótól, hogy hány méteres magasságot vizsgáljon!

Fájlok a kérészéletű adatok helyett

A szemfülesebbeknek bizonyára feltűnt már, hogy létezik „néhány” program, amelyik háttértárról képes betölteni, illetve háttértárra képes menteni az adatokat, amelyekkel dolgozik. Mi eddig ilyen programot nem tudtunk írni, de épp itt az ideje ezen változtatni!

Ha elég messziről tekintünk a számítógépeken tárolt fájlokra, akkor alapvetően kétféle különböztetünk meg. Az egyikben szöveg van, méghozzá formázatlanul, olyan, amely csak számokat, betűket, szóközöket, írásjeleket tartalmaz. Nincs benne dőlt betű, aláhúzás, nincs megadva betűtípus és -méret, azaz ezek a fájlok szigorúan nem olyan szöveget tartalmaznak, mint amilyet irodai szövegszerkesztő alkalmazással készítünk. Az ilyen fájlok kiterjesztése nagyon gyakran `.txt`, az angol text, azaz ’szöveg’ szó alapján. A `.txt` kiterjesztésűeken kívül ebbe a csoportba tartozik még többek között a táblázatos adatokat tárolni képes `.csv` fájltípus, valamint a weboldalak kódját tartalmazó `.html` és `.css` kiterjesztéssel bíró állomány- és még jó néhány fájltípus. Szöveget tartalmazó fájlokat készíthetünk az úgynevezett egyszerű szerkesztőkkel, például a Windows Jegyzettömbjével, de ilyen fájlt ír minden IDE, az is, amivel a digitáliskultúra-órákon szerkesztjük programjainkat.

A „másik” fájltípus az összes többi. Kiterjesztéseik és a bennük tárolt adatok rendkívül változatosak. Idetartoznak a lefordított programok, a videók, a képek, a hangok, az irodai szövegszerkesztők és táblázatkezelők saját formátumú állományai, a tömörített fájlok, és még ki tudja, hányféle fájl.

Könyvünk programozásról szóló részében csak a szövegfájlokkal foglalkozunk. Ennek a leckénknek pedig minden példájában az alábbi, `allatok.txt` fájlt használjuk, amely megtalálható a tankönyv weblapjáról letöltött fájlok között. A fájlban kedvenc tanyánk háziállatai vannak felsorolva. Érdemes tudni, hogy a fájl tizenöt soros, és Latyak kacsa sora után nem kezdünk új sort, nem nyomunk Entert. Az egyes sorokban találjuk az állat nevét, azt, hogy miféle állatról van szó (a későbbiekben az egyszerűség kedvéért sokszor és pontatlanul fajtának nevezzük ezt a tulajdonságot), és azt is, hogy ez az állat hány éves.

```
Totyak kacsa 1
Lakli kutya 12
Hektor kutya 4
Puha birka 3
Nyeldekel kecske 5
Csinibaba szarvasmarha 3
Nyakas liba 2
Dinka malac 1
Coca malac 1
Smafu malac 3
Tarajos kakas 1
Csillag macska 2
Zokni macska 3
Pamacs birka 4
Latyak kacsa 2
```

Szövegfájlok beolvasása

Az első programunk mindössze annyit tesz, hogy megnyitja a fájlt, és kiírja a sorait, majd zárja a fájlt.

Figyeljük meg, hogy az `open()` függvény paraméteréül csak a fájl nevét adtuk meg. Ha a fájl nem abban a mappában van, ahonnan a programot futtatjuk, vagy ha elírjuk a nevet, vagy ha egyáltalán nincs is ilyen fájl, akkor a programunk panaszkodik miatta.

```
C:\Users\raerek\programok>beolvas_es_kiir.py
Traceback (most recent call last):
  File "C:\Users\raerek\programok\beolvas_es_kiir.py", line 1, in <module>
    forrásfájl = open('elírtuk.txt')
FileNotFoundError: [Errno 2] No such file or directory: 'elírtuk.txt'
```

Bizonyára feltűnik az is, hogy amikor kiírja a programunk a fájl sorait – azaz sikeresen megtalálta a fájlt, megnyitotta, és olvasott belőle –, akkor is minden sor alatt egy üres sort is megjelenít. Miért van ez így?

Az ok az, hogy a szövegfájlok sorainak a végét egy Linuxon és macOS-en egy új sor, más néven *soremelés* (angolul: line feed, LF), Windowson pedig egy *kocsivissza* és egy *soremelés* karakter zárja. A *kocsi vissza* és a *soremelés* szavak az írógépek korából valók, és itt arra utalnak, hogy már a negyven-ötven évvel ezelőtt készült nyomtatók is a fájlok nyomtatásakor a *kocsi vissza* jel hatására vitték vissza a nyomtatófejet a sor elejére, és a *soremelés* karakter hatására igazították a lapot egy sorral lentebb.

A fájl sorainak végén tehát van egy vagy több olyan karakter, amelyekből tudni lehet, hogy új sor kezdődik. Ezt a karaktert a programunk lelkiismeretesen beolvassa és kiírja, azaz új sor kezdődik. Igen ám, de a `print()` függvény alapértelmezetten is új sort kezd minden sor kiírása után. Ez eredményezi az üres sorokat. Ha meg akarunk szabadulni tőlük, kényelmes lehet a `print()` függvény alapértelmezett sorvége jelét, az új sort író `'\n'` értéket felülírni egy üres karakterlánccal. Azaz a

```
print(sor, end='')
```

utasítást használni. A legtöbbször azonban nem kiírjuk, hanem tároljuk az adatainkat – például egy listában –, és ilyenkor a szövegfájl sorainak végén lévő jel, jelek folyamatos gondot jelentenek. Szerencsére a Pythonban létezik egy olyan tagfüggvény, amely a szövegek végén lévő, számunkra gondot okozó végződést eltávolítja. A tagfüggvény neve `strip()`, azaz: *lecsupaszít*. Próbáljuk a kódunk 3. sorát így átírni:

```
3. print(sor.strip())
```

Látjuk, hogy mostanra nincsenek üres soraink a kiíráskor. A `strip()` tagfüggvény elég ügyes: figyel arra, hogy CR karakter vagy CR és LF karakterpár zárja a sort, és azt veszi le, amit kell.

Ha a fájlban lévő adatokat tárolni szeretnénk, akkor egy nagyon hasonló programot írunk:

```
1. állatok = []
2.
3. forrásfájl = open('allatok.txt')
4. for sor in forrásfájl:
5.     sor = sor.strip()
6.     állatok.append(sor)
7. forrásfájl.close()
8.
9. print(állatok)
```

Üres lista, ebbe pakolunk majd.

Nyitjuk a fájlt.

Lecsupaszítjuk a sort.

A csupasz sort a lista végére fűzzük.

Zárjuk a fájlt.

Kiírjuk a listánkat.

A fájlnyitás, -feldolgozás, -zárás hármassága a legtöbb programozási nyelven a fentiekhez hasonlóan zajlik. A Python fájlműveletekhez hasonló másik, egyre elterjedtebb módszere az, hogy a fájlkezelést végző részt egy `with` utasítással kezdődő programrész belsejében helyezzük el. Ilyenkor a fájl zárására nincs szükség, mert a `with` belsejét elhagyva a Python automatikusan zárja a fájlt. A fenti program 3–7. sorait cseréljük az alábbi négyre:

```
with open('allatok.txt') as forrásfájl:
    for sor in forrásfájl:
        sor = sor.strip()
        állatok.append(sor)
```

A with belsejében vagyunk, itt nyitva van a fájl.

Használjuk a két módszer közül azt, amelyik jobban kézre áll.

Sorokat már be tudunk olvasni, de a legtöbbször szeretnénk a bennük lévő adatokat szétszedve látni – a mostani példában külön az állat nevét, fajtáját és korát. Listák szétdarabolására, hasítására a `split()` tagfüggvény való. A működését az alábbi program szemlélteti. Próbáljuk ki a programot!

```
1. mondat = input('Írj ide egy mondatot! ')
2. szavak = mondat.split()
3. print('A szavakat tartalmazó lista:', szavak)
4. print('Újra összerakva:')
5. print(' '.join(szavak))
```

A split listává alakítja a karakterláncot.

A `split()` tagfüggvény alapértelmezetten szóközöknél, tabulátoroknál és hasonló karaktereknél (angol összefoglaló nevük: whitespace) hasít. Ha nekünk például pontosvessző jelzi az egyes adatok határát, akkor a `split(';')` formában használjuk az utasítást.

Az állatos programunkat az 5. sorban bővítsük a sor végére írt `split()` tagfüggvénnyel! Futtassuk a programot, és figyeljük meg a kimenetét: egy kétdimenziós listát látunk.

```
1. állatok = []
2.
3. forrásfájl = open('allatok.txt')
4. for sor in forrásfájl:
5.     sor = sor.strip().split()
6.     állatok.append(sor)
7. forrásfájl.close()
8.
9. print(állatok)
```

A sor karakterláncot először lecsupaszítjuk, majd darabjaira szedjük. Az eredmény egy lista, amelyet visszateszünk a sor nevű változóba.

A sor nevű listát az „állatok" nevű lista végére tesszük. Az „állatok" egy kétdimenziós lista.

Figyeljük meg, hogy a beolvasott számokat – az állatok korát – egyelőre szövegként tároljuk. Legcélszerűbb talán még a beolvasás során számmá alakítanunk őket. Ezt megtehetjük a fenti kód 5. és 6. sora közé szúrt

```
sor[2] = int(sor[2])
```

utasítással.

Természetesen semmi nem kötelez bennünket arra, hogy kétdimenziós listába töltsük az adatainkat. Ha a fenti kód 5. és 6. sora közé ezt az utasítást szúrjuk:

```
szótár = {'név': sor[0], 'fajta': sor[1], 'kor': int(sor[2])}
```

illetve az *állatok* lista végére nem a sorlistát, hanem a belőle képzett `szótár` nevű szótárat illesztjük a jelenlegi 6., a módosítást követően 7. sorba, akkor meg is oldottuk a feladatot.

A típusalgoritmusainkat természetesen a fájlokból beolvasott adatokkal is tudjuk használni. Ha például a beolvasást követően ki szeretnénk írni a legöregebb állat nevét és fajtáját, akkor a szótárakat használó változatot nagyjából hasonlóan kell kiegészítenünk:

A „legöregebb" az egész szótárat tárolja, nem csak az állat korát. Kezdetben a legelső állatot tekintjük a legöregebbnek.

```
12. legöregebb = állatok[0]
13. for állat in állatok:
14.     if állat['kor'] > legöregebb['kor']:
15.         legöregebb = állat
16.
17. print(legöregebb['név'], legöregebb['fajta'])
```

Ha találunk öregebb állatot, akkor tároljuk.

Szövegfájlok írása

A szövegfájlok írása nem sokkal bonyolultabb az olvasásuknál. A fájl megnyitását ezúttal is az `open()` függvény végzi. A függvény elhagyható paramétere a megnyitás módját jelzi. Ennek alapértelmezett értéke az `'r'`, azaz olvas (angolul: read), és ezt eddig nem írtuk ki soha, mert olvasni akartunk a fájlból. Ha írni akarunk egy fájlba, akkor az `open()` függvény a `'w'` (write, azaz ír) paraméter hatására új fájlt kezd a megadott néven – ha volt már ilyen, azt felülírja. Az `'a'` (append: hozzáfűz) paramétert megadva pedig az esetleg meglévő fájlt folytathatjuk.

Megnyitottuk hát a fájlunkat, de még írni is kell bele. Mindössze annyi a dolgunk, hogy a `print()` függvénynek átadjuk azt a fájlobjektumot, amibe írni kell.

Módosítsuk úgy az előző programunkat, hogy a legöregebb állat nevét ne a képernyőre, hanem az `oreg.txt` állományba írja! Mindössze az előző programunk utolsó sorát kell egy kicsit kiegészítenünk, illetve elé és mögé beírni egy-egy újat:

```
17. célfájl = open('oreg.txt', 'w')
18. print(legöregebb['név'], legöregebb['fajta'], file=célfájl)
19. célfájl.close()
```

Írásra nyitjuk meg a fájlt.

Ebbe a fájlba írjon a print().

Az írásra megnyitott fájlt mindenképp zárjuk be!

Most, hogy már úgy írunk és olvasunk fájlokat, mintha mindig is remekül ment volna, ki kell térnünk egy fontos részletre. A számítógép a karaktereket számkóddal tárolja, mind a memóriában, mind a fájlokban. A magyar nyelv ékezetes karaktereinek számmá alakítására sokféle módszer van. Az átalakítás mostanra jó pár éve szinte mindenhol elterjedt szabványa az UTF-8. 2021-ben, e sorok születése idején egyes statisztikák szerint a világháló tartalmának 97 százaléka így van kódolva. A mai Linuxok és macOS-ek is ezt használják.

Nos, a Windows parancssora kivétel, és a Windowson futó Python ezt követi. Ha azt szeretnénk, hogy az ékezetes karaktereket tartalmazó, általában UTF-8 kódolással mentett szövegfájljainkat a Pythonban írt programjaink Windowson hiba nélkül olvassák és írják, akkor tennünk kell ennek érdekében.

A helyzetet megoldja, ha a PYTHONUTF8 környezeti változót 1-re állítjuk. Ezt legkönnyebben a parancssorban kiadott `set PYTHONUTF8=1` paranccsal tehetjük meg. A módszer hátránya, hogy minden parancssori ablak megnyitásakor újra ki kell adni a parancsot. A környezeti változók tartósabb megadására szolgáló módszert találhatunk az interneten.

A másik megoldás, ha programjainkban az `open()` függvény argumentumai között szerepeltetjük, hogy `encoding='utf-8'`, íráskor és olvasáskor egyaránt. Így a programjaink Windowson gond nélkül lefutnak, Linuxon és macOS-en pedig nem lesz változás, azaz ott is gond nélkül lefutnak.

Minderre nincs szükség, ha a szövegfájlunk nem tartalmaz ékezetes karaktert, épp ezért nincs ékezet az `allatok.txt` fájlban lévő állatok nevében és fajtájában sem.

A szövegfájlok beolvasásának egyéb módszerei

A szövegfájlok soronkénti beolvasása az esetek nagy többségében megfelel a munkánkhoz. Ha mégsem így volna, az alábbiakban közöljük még a szövegfájlok beolvasásának néhány egyéb módszerét.

Beolvasás egy lépésben egy listába: a lista sorainak a végén ott vannak a sorvége jelek is, amelyeket nekünk kell leszedni utólag:

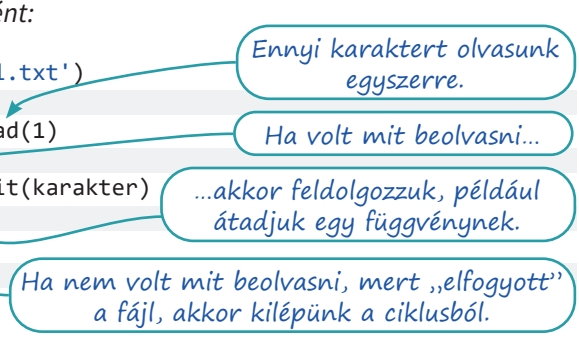
```
1. forrásfájl = open('szovegfajl.txt')
2. lista = forrásfájl.readlines()
3. forrásfájl.close()
```

Beolvasás egy lépésben egyetlen szövegként: ritkán használjuk, de az ezt követő módszer megértéséhez szükség van rá:

```
1. forrásfájl = open('szovegfajl.txt')
2. szöveg = forrásfájl.read()
3. forrásfájl.close()
```

Beolvasás egy vagy néhány karakterenként:

```
1. forrásfájl = open('szovegfajl.txt')
2. while True:
3.     karakter = forrásfájl.read(1)
4.     if karakter:
5.         csinálunk_vele_valamit(karakter)
6.     else:
7.         break
8. forrásfájl.close()
```



Kópiakészítés és digitális Hamupipőke – Másolunk, kiválogatunk és szétválogatunk típusalgoritmussal

Tizedik évfolyamon megismerkedtünk a típusalgoritmusok – ha úgy tetszik, programozási tételek – egy csoportjával, mostanra pedig fel is elevenítettük használatukat. Az eddig megismertek az „egyszerű” típusalgoritmusok. Közös tulajdonságuk, hogy egy bejárható objektum sok eleméhez egyetlen értéket – az összegüket, az átlagukat, a legnagyobbat, egy kiválasztott elem értékét, egy logikai értéket – rendelünk velük.

E mostani leckével indulóan olyan típusalgoritmusokat ismerünk meg, amelyek egy értéksorozathoz, bejárható objektumhoz egy másik bejárható objektumot rendelnek. Ezek az **összetett típusalgoritmusok.**

Másolás

A másolás típusalgoritmusának lényege a következő:

1. járjuk be egy bejárható objektum elemeit,
2. valamilyen módon alakítsuk át az egyes elemeket, és az átalakítást követően
3. gyűjtsük újabb bejárható objektumba az így kapott elemeket.

Az elemek átalakítását matematikusmegfogalmazással mondhatjuk úgy is, hogy a bejárható objektum elemeihez valamilyen hozzárendelési szabállyal, hozzárendelő függvénynyel másik elemet rendelünk.

Mindez mondatszerű leírásban:

```
sorozat = valamilyen bejárható objektum
másolat = üres bejárható objektum
ciklus a sorozat minden elem-ére
    átalakított_elem = hozzárendelő_függvény(elem)
    másolat-hoz hozzáfűz(átalakított_elem)
ciklus vége
```

Megjegyezzük, hogy amikor a „hozzárendelő függvény” feladata kellőképp egyszerű, nem feltétlen írunk külön függvényt, hanem helyben kiszámítjuk a hozzárendelés értékét.

Az egyik leggyakoribb másolási feladat a nagyon gyakran előforduló típusátalakítás, például amikor karakterekként tárolt számokat alakítunk „igazi” (egész vagy lebegőpontos) számokká, vagy fordítva. Három módszert is bemutatunk.

Az első a leghagyományosabb, teljesen megfelel a fenti mondatszerű leírásnak.

A másodikban szereplő `map()` függvénynek két paramétere van: az első egy függvény, a második egy bejárható objektum. A `map()` a paraméterként megadott függvényt a bejárható objektum minden elemén lefuttatja. A `map()` visszatérési értékét a `list()` függvénnyel listává alakítjuk. Azért közöljük ezt a módszert, mert Python nyelven ez a legrövidebb megfogalmazása a másolás algoritmusának.

A harmadik módszer a listaértelmezés (angolul: list comprehension) módszere. Lényegében ugyanaz, mint az első, csak egy sorba átírva. Python nyelvű kódokat keresgélve nagyon gyakran találkozunk ezzel a módszerrel az interneten.

```
1. számok_string_formában = ['1', '5', '2', '3', '4']
2. print('Stringként: ', számok_string_formában)
3.
4. # első módszer: ciklus
5. számok = []
6. for string in számok_string_formában:
7.     szám = int(string)
8.     számok.append(szám)
9. print('Első módszerrel: ', számok)
10.
11. # második módszer: map
12. számok = list(map(int, számok_string_formában))
13. print('Második módszerrel: ', számok)
14.
15. # harmadik módszer: listaértelmezés
16. számok = [ int(string) for string in számok_string_formában ]
17. print('Harmadik módszerrel: ', számok)
```

Most az int() függvény a mondatszerű leírásban emlegetett „hozzárendelő" függvény.

A map() az itt megadott függvényt futtatja le...

...ennek az objektumnak minden elemén.

Ez a két forma ugyanazt csinálja.

15. példa: A taxis bevételei

Adott a tizedik évfolyamos tankönyvből ismerős taxisunk piculában kifejezett bevételeinek listája. A taxis adóterhei jelentősek: mire kifizeti a mindenféle adókat, a bevétel egészre kerekített 49 százalékáról lemondhat. Adjuk meg a taxis adózott bevételeinek a listáját!

```
1. bevételek = [1, 5, 2, 3, 4]
2. adózott_bevételek = []
3.
4. for bevétel in bevételek:
5.     adó = round(bevétel * 0.49)
6.     marad = bevétel - adó
7.     adózott_bevételek.append(marad)
8.
9. print('A taxis adózott bevételei:', ', '.join(map(str, adózott_bevételek)))
```

Most a round() és a szorzás a „hozzárendelő" függvény.

Itt meg valójában végzünk egy másik másolást.

16. példa: Libatömegek farkas előtt és farkas után

Az a szituáció is ismerős lehet a tizedik évfolyamos kötetből, amikor 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. Adjuk meg azt a listát, amelyik a farkassal való találkozást követő libatömegeket tartalmazza! Amelyik libát a farkas elvette, annak helyére írjunk nullát!

```
1. def farkas(liba_tömege):
2.     if liba_tömege > 3:
3.         return 0
4.     else:
5.         return liba_tömege
6.
7. libák = [1, 5, 2, 3, 4]
8. róka_libái = []
```

```
 9.
10. for liba in libák:
11.     marad = farkas(liba)
12.     róka_libái.append(marad)
13.
14. print('A rókánál maradt libák:', ', '.join(map(str, róka_libái)))
```

17. példa: A tanya állathangjai

Az előző leckéből már ismert `allatok.txt` fájl tartalmazza tanyánk állatait. Minden állatfajnak ismert a hangja (például macska: nyaú, malac: röf). Állítsuk össze azt a listát, amely megmutatja, hogy milyen hangokkal köszöntenek bennünket állataink, amikor hazaérünk a tanyára! Feltételezzük, hogy az állatok a fájlban található sorrendjüknek megfelelően szólalnak meg.

```
 1. állathangok = {'kacsa': 'háp', 'kutya': 'vaú',
 2.                'birka': 'bee', 'kecske': 'mek',
 3.                'szarvasmarha': 'mú', 'liba': 'gá',
 4.                'malac': 'röf', 'kakas': 'qqriq',
 5.                'macska': 'nyaú'}
 6.
 7. köszöntések = []
 8.
 9. forrásfájl = open('allatok.txt')
10. for sor in forrásfájl:
11.     fajta = sor.strip().split()[1]
12.     köszöntések.append(állathangok[fajta])
13.
14. forrásfájl.close()
15.
16. print('Állataink üdvözlete: ', ', '.join(köszöntések))
```

Kiválogatás és szétválogatás

A kiválogatás típusalgoritmusa majdnem olyan, mint a másolásé. Ezúttal nem alakítjuk át az elemeket, hanem az eredetieket másoljuk, de nem mindet, hanem csak azokat, amelyek megfelelnek valamilyen feltételnek.

Mindez mondatszerű leírásban:

```
sorozat = valamilyen bejárható objektum
kiválasztottak = üres bejárható objektum
ciklus a sorozat minden elem-ére
   ha elem adott tulajdonságú, akkor:
      kiválasztottak-hoz hozzáfűz(elem)
ciklus vége
```

A szétválogatás típusalgoritmusa abban különbözik a kiválogatásétól, hogy több listába vagy egyéb objektumba válogatjuk szét az elemeket. Az egyikbe kerülnek azok, amelyek megfeleltek a feltételnek, a másikba azok, amelyek nem. Az is elképzelhető, hogy többfelé

válogatjuk szét az eredeti elemsorozatunkat: az egyik gyűjteménybe kerülnek a kicsik, a másikba a közepesek, a harmadikba a nagyok.

18. példa: A farkas és a róka libalakomája

Az 16. példában szereplő libatolvajlások ismeretében adjuk meg a két ragadozó szárnyasainak listáit! A listák felhasználásával állapítsuk meg, hogy melyik ragadozónak hány darab, illetve hány kilónyi liba jutott! A kiírást végeztessük eljárással, melynek paraméterei a ragadozó neve és a ragadozó tyúkjait tartalmazó lista!

```
1. def kiír(ragadozó, libalista):
2.     print('A', ragadozó, 'libái:',
3.         ', '.join(map(str, libalista)))
4.     print('A libák száma:', len(libalista),
5.         'össztömege:', sum(libalista), 'kg.')
6.
7. libák = [1, 5, 2, 3, 4]
8. róka_libái = []
9. farkas_libái = []
10.
11. for liba in libák:
12.     if liba <= 3:
13.         róka_libái.append(liba)
14.     else:
15.         farkas_libái.append(liba)
16.
17. kiír('róka', róka_libái)
18. kiír('farkas', farkas_libái)
```

A join() csak stringeket tud összefűzni, szóval konvertálunk.

megszámlálás

sorozatszámítás/összegzés

Szétválogatás: a nagy libák a farkasé, a kicsik a rókáé.

Az eljárást kétszer hívjuk.

19. példa: Hőségriadós napok

A hőségriasztás legalacsonyabb fokozata arról tájékoztat, hogy egy napon 25 °C-ot meghaladó hőmérséklet várható. A következő heti előrejelzés szótárban tárolt adatai alapján gyűjtsük listába, és írjuk a képernyőre azokat a napokat, amikor hőségriasztást kell kiadni!

```
1. előrejelzések = {'hétfő': 19, 'kedd': 23, 'szerda': 26,
2.                  'csütörtök': 27, 'péntek': 19,
3.                  'szombat': 18, 'vasárnap': 18}
4.
5. hőségriadós_napok = []
6. for nap in előrejelzések:
7.     if előrejelzések[nap] > 25:
8.         hőségriadós_napok.append(nap)
9.
10. print('Hőségriadós napok:', ', '.join(hőségriadós_napok))
```

A szótárat kulcsok szerint járjuk be. Írhatnánk azt is, hogy előrejelzések.keys().

kiválogatás

Kópiakészítés és digitális Hamupipőke – Az eddig tanult összetett típusalgoritmusok a gyakorlatban

20. példa: Gyalogtúra

A könyv webhelyéről letöltött `tura.txt` állományban egy gyalogtúrán hárompercenként rögzített magassági adatokat találunk. Olvassuk be a fájlt, majd állítsunk elő a felhasználásával egy olyan listát, amelyik azt mutatja, hogy „Fel” vagy „Le” változott-e a túrázónál lévő GPS-készülék által mért magasság az előző mérési pont óta, esetleg megegyezik az előzővel („=”)! Írjuk az új lista elemeit vesszővel elválasztva a képernyőre!

Írjuk meg a fájlbeolvasást követő részt mondatszerű leírással! Minthogy (majdnem) minden adatnak megfeleltetünk egy újat, a másolás típusalgoritmusát kell használnunk.

```
Program szintmérés:
magasságok := fájlból beolvasott adatok bejárható objektumban
irányok := üres bejárható objektum
ciklus i = 1-től (magasságok elemszáma)-1 -ig
   ha magasságok[i] < magasságok[i-1], akkor
     irányok := irányok + „Le”
   különbenha magasságok[i] > magasságok[i-1], akkor
     irányok := irányok + „Fel”
   különben
     irányok := irányok + „=”
ciklus vége
Program vége.
```

Kódoljuk a programot, és mentsük `szintmeres.py` néven!

```
1. with open('tura.txt') as forrásfájl:
2.     magasságok = forrásfájl.read().strip().split(', ')
3.
4. magasságok = list(map(int, magasságok))
5.
6. irányok = []
7. for index in range(1, len(magasságok)):
8.     if magasságok[index] < magasságok[index-1]:
9.         irányok.append('Le')
10.    elif magasságok[index] > magasságok[index-1]:
11.        irányok.append('Fel')
12.    else:
13.        irányok.append('=')
14.
15. print('A magasság változása:', ', '.join(irányok))
```

Megnyitjuk a fájlt.

Beolvassuk az első (és egyetlen) sorát.

Levesszük a sorvége jel(ek)et.

Vessző és szóköz mentén daraboljuk a sort.

Egésszé alakítjuk a „magasságok” lista minden egyes értékét.

Megvalósítjuk a fenti pszeudokódot.

Módosítsuk az előző programot úgy, hogy a három métert meg nem haladó változásokat még egyenlőnek tekintse! Mindössze a fenti kód 8. és 10. sorában kell egy keveset változtatnunk: a kettőspont elé kell odaírni, hogy „-3”, illetve „+3”.

21. példa: E terkes mecske leberetette e tejfelt

Lehet egy egyperces rettenet, melyet eszperente nyelven egy ember nyelvel? Az `eszperente.py` program egy hangyányit primitív lesz, ugyanis a megadott mondatból úgy ír eszperentét, hogy minden magánhangzót *e*-re cserél, például:

A torkos macska leborította a tejfölt. *E terkes mecske leberetette e tejfelt.*

Az első verzió elég, ha csupa kisbetűs mondatokat kezel, aztán oldjuk meg, hogy a nagybetűket is tudja kezelni! Minthogy minden karakternek megfeleltetünk egy másikat, ismét a másolás típusalgoritmusa fog segíteni.

```
1. magyar_mondat = input('Kérek egy mondatot! ')
2. eszperente_mondat = []
3.
4. magánhangzók = 'aáeéiíoóöőuúüű'
5.
6. for karakter in magyar_mondat:
7.     if karakter in magánhangzók:
8.         eszperente_mondat.append('e')
9.     else:
10.        eszperente_mondat.append(karakter)
11.
12. print('Gyenge eszperente:', ''.join(eszperente_mondat))
```

Be kell-e ide írni az „e” betűt?

Eldöntés a másolás kellős közepén. Mit dönt el?

A nagybetűket is kezelő változat:

```
6. for karakter in magyar_mondat:
7.     if karakter.lower() in magánhangzók:
8.         if karakter.isupper():
9.             eszperente_mondat.append('E')
10.        else:
11.            eszperente_mondat.append('e')
12.    else:
13.        eszperente_mondat.append(karakter)
```

A kisbetűs változat alapján döntünk.

Nagy magánhangzó esetén nagy E kerül az eszperente változatba…

…kicsi esetén kis e.

22. példa: Kutya- és macskaoltások

Kutyáinkat és macskáinkat be szeretnénk oltatni. Mindegyiknek adatnánk veszettség elleni oltást, és a kutyáknak parvovírus ellenit is. A már használt `allatok.txt` fájlból gyűjtsük az oltás nevének megfelelő listába azoknak az állatoknak a nevét, amelyek az adott oltást kapni fogják! Jelenítsük meg a listák tartalmát!

Ki kell válogatnunk, hogy mely állatok tartoznak az egyik, illetve a másik fajhoz. Ehhez a kiválogatás típusalgoritmusát használjuk.

Írjuk meg mondatszerű leírással a két oltási listát kialakító részt! Feltételezhetjük, hogy az állatokat kétdimenziós listában tároltuk a fájl beolvasását követően.

```
Program kutyamacska:
veszettség := üres gyűjteményes adatszerkezet
parvo := üres gyűjteményes adatszerkezet
ciklus állatok minden állat-jára
  ha állat[1] = „kutya” vagy állat[1] = „macska” akkor
```

```
    veszettség-et bővítjük állat[0]-val
  ha állat[1] = „kutya” akkor
    parvo-t bővítjük állat[0]-val
ciklus vége
Program vége.
```

Kódoljuk a programot `kutyamacska.py` nevű állományba!

```
 1. állatok = []
 2.
 3. with open('allatok.txt') as forrásfájl:
 4.     for sor in forrásfájl:
 5.         sor = sor.strip().split()
 6.         sor[-1] = int(sor[-1])
 7.         állatok.append(sor)
 8.
 9. veszettség = []
10. parvo = []
11.
12. for állat in állatok:
13.     if állat[1] == 'kutya' or állat[1] == 'macska':
14.         veszettség.append(állat[0])
15.     if állat[1] == 'kutya':
16.         parvo.append(állat[0])
17.
18. print('Veszettség ellen kap oltást:', ', '.join(veszettség))
19. print('Parvovírus ellen kap oltást:', ', '.join(parvo))
```

Az utolsó adat szám, számként is tároljuk. Még akkor is rendesen tároljuk az adatainkat, ha most épp nem lesz rájuk szükség.

23. példa: Tojásrakók

Van egy listánk arról, hogy a háziállataink közül melyik faj egyedei raknak tojást. A már használt `allatok.txt` fájlból* gyűjtsük ki azoknak a nevét, amelyeknél elképzelhető ilyen esemény! Az állatok nevéből ezúttal ne következtessünk a nemükre!

Ki kell válogatnunk a tojásrakókat – ezt a típusalgoritmust használjuk a `tojasrakok.py` programban.

A kód eleje megegyezhet az előző feladat megoldásáéval, így csak a 9. sortól kezdődő részt adjuk meg itt.

```
 9. tojásrakó_fajok = ['kacsa', 'liba', 'kakas']
10. tojásrakók_nevei = []
11.
12. for állat in állatok:
13.     if állat[1] in tojásrakó_fajok:
14.         tojásrakók_nevei.append(állat[0])
15.
16. print('Tojásrakó fajhoz tartozik:', ', '.join(tojásrakók_nevei))
```

*Tarajos fajként a fájlunkban „kakas” szerepel, aminek az az oka, hogy a „házi tyúk” szóban szerepel ékezetes karakter, és ez – mint ahogy láttuk – bonyolíthatja a programot.

Kópiakészítés és digitális Hamupipőke – Az eddig tanult összetett típusalgoritmusok újabb gyakorlatokban

24. példa: Jók és rosszak

George Orwell *Állatfarm* című regényében egy tanya állatai fellázadnak gonosz gazdájuk, illetve általában az emberek ellen. A legegyszerűbb gondolkodásúak számára is világossá óhajtották tenni az új világrendet, így a lázadás vezetői kiadták a „Négy láb jó, két láb rossz!” jelszót. A szárnyasok rámutattak, hogy ez a szlogen számukra kirekesztő, mire a lázadás ideológusai elmagyarázták, hogy ebben a kontextusban a madarak szárnya is lábnak minősül. Nincs hát szó a szárnyasok megbélyegzéséről, a jelszó az emberi lényeket minősíti.

Írjunk programot az előző leckében megírt program átalakításával `orwell.py` néven, amely a jelszónak még az említett utólagos korrekciója előtt, a jelszó szigorúan vett jelentése szerint kategorizálja állatainkat! Olvassuk be soronként az `allatok.txt` fájlt, és minden egyes sor beolvasását követően helyezzük a `jók`, illetve a `rosszak` listába a sorban szereplő állat nevét az előző feladatban is segítségünkre lévő lista felhasználásával! Jelenítsük meg vesszővel elválaszott felsorolásként az egyes listák tagjait!

A program minden állatot megtart, elhelyez egy gyűjteményes adatszerkezetben, azaz a szétválogatás típusalgoritmusát használjuk.

```
állatok = []

with open('allatok.txt') as forrásfájl:
    for sor in forrásfájl:
        sor = sor.strip().split()
        sor[-1] = int(sor[-1])
        állatok.append(sor)

tojásrakó_fajok = ['kacsa', 'liba', 'kakas']
jók = []
rosszak = []
for állat in állatok:
    if állat[1] in tojásrakó_fajok:
        rosszak.append(állat[0])
    else:
        jók.append(állat[0])

print('Rosszak:', ', '.join(rosszak))
print('Jók:', ', '.join(jók))
```

szétválogatás

Ha elkészültünk, bővítsük a programot úgy, hogy az egyes sorok beolvasását követően adjon szövegesen megfogalmazott véleményt is, például:

```
Totyak kacsa kétlábú, azaz rossz.
```

Úgy látjuk, hogy a véleményalkotást és a vélemény megjelenítését már megéri függvénybe kiszerveznünk. Írjunk hát olyan függvényt,

- melynek paramétere egy szótár, amely a függvénynek átadott állat nevét, faját és korát tartalmazza;
- amelynek helyi, csak a függvény belsejében létező adatszerkezete a tojásrakó fajokat felsoroló, a döntéshozatal alapjául szolgáló lista;
- amelyik a fenti mintának megfelelő véleményezést ír a képernyőre és
- amelyik visszatérési értéke az adott állat „jóságát” jelentő logikai érték!

A `jók` és a `rosszak` listát a programunk új változata az előbb elkészített függvény használatával töltse fel!

```
 1. def jó(állat):
 2.     tojásrakó_fajok = ['kacsa', 'liba', 'kakas']
 3.     if állat['faj'] in tojásrakó_fajok:
 4.         print(állat['név'], állat['faj'], 'kétlábú, tehát rossz.')
 5.         return False
 6.     else:
 7.         print(állat['név'], állat['faj'], 'négylábú, tehát jó.')
 8.         return True
 9.
10. állatok = []
11.
12. with open('allatok.txt') as forrásfájl:
13.     for sor in forrásfájl:
14.         sor = sor.strip().split()
15.         állat = {'név': sor[0], 'faj': sor[1], 'kor': int(sor[2])}
16.         állatok.append(állat)
17.
18. jók = []
19. rosszak = []
20. for állat in állatok:
21.     if jó(állat):
22.         jók.append(állat['név'])
23.     else:
24.         rosszak.append(állat['név'])
25.
26. print('Rosszak:', ', '.join(rosszak))
27. print('Jók:', ', '.join(jók))
```

egy állat = egy szótár

Itt hívjuk a függvényt.

A szétválogatás a függvény visszatérési értéke alapján történik.

25. példa: Hajónapló

Adott egy fájl `hajonaplo.txt` néven, melyet a tankönyv weblapjáról letöltött fájlok között találunk. A fájl soronként két, kötőjellel elválasztott számot tartalmaz. Az első szám minden sorban azt mutatja, hogy hány fokos irányba haladt a hajó, a második azt, hogy hány tengeri mérföldet haladt abba az irányba. A `kormanyos.py` programban az a feladatunk, hogy megadjunk egy olyan listát, amely azt sorolja fel, hogy a hajót az egyes irányváltások – az egyes sorok – között jobbra vagy balra kormányozták-e. Mindig arra kormányozzák a hajót, amerre kisebbet kell fordulnia. Ha pontosan száznyolcvan fokot kellene fordulni, akkor a kormányos véletlenszerűen dönti el, hogy jobbra vagy balra fordul-e.

Majdnem minden adatnak megfeleltetünk egy újat, azaz a másolás típusalgoritmusát kell használnunk.

A fordulás irányát meghatározó algoritmus lehet például a következő:

```
ha (új_irány - régi_irány + 360) mod 360 < 180 akkor
    ki: „J”
különbenha (új_irány - régi_irány + 360) mod 360 > 180, akkor
    ki: „B”
különben:
    ki: „J” és „B” közül valamelyik véletlenszerűen
```

Írjunk a fenti mondatszerű leírásból függvényt, amelynek két paramétere a régi és az új irány, a visszatérési értéke pedig egy J vagy egy B karakter! Írjuk meg a főprogramot, amely beolvassa a `hajonaplo.txt` fájlt, és a kormányzásokat a `kormanyzasok.txt` fájlba írja! A kimeneti fájl egy-egy sora tartalmazza a kormánymozdulat előtti útirányt, a kormánymozdulat irányának betűjét és a kormánymozdulatot követő irányt, szóközzel elválasztva. Ha a bemeneti fájl adatai:

```
107-12
109-34
210-87
202-3
349-198
17-251
197-37
```

akkor a kimeneti fájléi:

```
107 J 109
109 J 210
210 B 202
202 J 349
349 J 17
17 J 197
```

```
1. import random
2.
3. def irány(régi_irány, új_irány):
4.     if ((új_irány - régi_irány + 360) % 360 < 180):
5.         return 'J'
6.     elif ((új_irány - régi_irány + 360) % 360 > 180):
7.         return 'B'
8.     else:
9.         return random.choice(['B', 'J'])
10.
11. napló = []
12. with open('hajonaplo.txt') as forrásfájl:
13.     for sor in forrásfájl:
14.         sor = sor.strip().split('-')
15.         sor = list(map(int, sor))
```

Úgy szokás, hogy az importálást követően adjuk meg a függvényeinket, eljárásainkat.

az előző oldalon olvasható pszeudokód megvalósítása

darabolás a kötőjel mentén

```
16.         napló.append(sor)
17.
18. with open('kormanyzasok.txt', 'w') as célfájl:
19.     for index in range(len(napló)-1):
20.         print(napló[index][0],
21.             irány(napló[index][0], napló[index+1][0]),
22.             napló[index+1][0],
23.             file = célfájl)
```

Ezúttal mindenképp át kell alakítani a számokat tartalmazó karakterláncokat!

26. példa: Távirat

2021. április 29-én lehetett utoljára táviratot feladni a magyar postahivatalokban. Ekkor már régen nem morzekóddal továbbították a jeleket.

Volt idő, amikor a morzekódokkal csak az angol ábécé (nagy)betűit és a számjegyeket lehetett kódolni, így a magyar ékezetes betűk helyén több betűből álló összetételeket küldtek. Az *á*-ból *aa,* az *é*-ből *ee* lett, és volt még *ii, oo, uu.* Az *ö* helyett *oe,* az *ő* helyett *ooe,* az *ü* helyett *ue,* az *ű* helyett *uue* került a szövegbe. A mondatvégi írásjelek helyett a STOP karaktersor morzekódját küldték át az éteren vagy a kábelen, a mondat belsejében lévőkről pedig lemondtak.

A „Förgeteg közeledik, előz bennünket!” mondat távirati alakja, ahogy a postás a távíróval elküldte, és ahogy a címzett olvashatta, a következő volt: „FOERGETEG KOEZELEDIK ELUUEZ BENNUENKET STOP”. Az „Áá, dehogy!” pedig ezt a formát öltötte: „AAAA DEHOGY STOP”.

Írjunk programot `tavirat.py` néven, amely a felhasználótól bekért szöveget a fent jelzett formára alakítva írja vissza a képernyőre!

Programunkban nem minden karakternek feleltetünk meg valamit, azaz lényegében kiválogatjuk azokat, amelyeknek lesz megfelelőjük. Ezt egy másolás követi: az ékezetmentes karakterek és a szóközök helyett saját magukat másoljuk vissza, az ékezetesek helyett a nekik megfelelő összetételt, a mondatvégi írásjelek helyett pedig szóközt és a STOP karaktersort.

A feladat ugyanakkor felfogható egyetlen másolásnak is: a mondat belsejében lévő írásjelek helyére üres karakterláncot, üres karaktert másolunk.

Ha az általunk használt programozási nyelv kínál egyszerű módot szövegek nagybetűssé alakítására, akkor használjuk azt! Ha nem, akkor ezt egy újabb másolással kell megoldanunk. A nagybetűssé alakításnak meg kell előznie a távirati szöveggé való alakítást.

A program lényegi és jól elkülöníthető része az átírt szövegváltozat előállítása. Szervezzük ezt ki függvénybe, amelynek paramétere a nagybetűs karakterlánc, visszatérési értéke pedig az átalakított karakterlánc! Az átalakítási szabályokat csak a függvénynek kell ismernie, a megfeleltetéseket tároló adatszerkezetet helyezzük el a függvény belsejében!

A feladatot először úgy oldjuk meg, hogy nagyon kevés, a modern nyelvekre jellemző módszert használunk:

```
1. def táviratttá_alakít(szöveg):
2.     szövegközi_írásjelek = ',;:"'
3.     írásjeltelen = []
4.     for karakter in szöveg:
5.         if karakter not in szövegközi_írásjelek:
6.             írásjeltelen.append(karakter)
7.     írásjeltelen = ''.join(írásjeltelen)
```

KIVÁLOGATjuk azokat a karaktereket, amelyeket megtartunk.

```
 8.
 9.     helyettesítések = [['Á', 'AA'], ['É', 'EE'], ['Í', 'II'],
10.                        ['Ó', 'OO'], ['Ú', 'UU'], ['Ö', 'OE'],
11.                        ['Ő', 'OOE'], ['Ü', 'UE'], ['Ű', 'UUE'],
12.                        ['.', ' STOP'], ['?', ' STOP'], ['!', ' STOP']]
13.     távirat = []
14.     for karakter in írásjeltelen:
15.         volt_helyettesítés = False
16.         for helyettesítés in helyettesítések:
17.             if karakter == helyettesítés[0]:
18.                 távirat.append(helyettesítés[1])
19.                 volt_helyettesítés = True
20.                 break
21.         if not volt_helyettesítés:
22.             távirat.append(karakter)
23.     távirat = ''.join(távirat)
24.     return távirat
25.
26. átalakítandó = input('Mi lesz a távirat szövege? ')
27. print('Táviratként:', távirattá_alakít(átalakítandó.upper()))
```

Ha szükséges cserélnünk, akkor a helyettesített karaktert másoljuk a távirat végére...

...különben az eredetit.

A nagybetűs üzenet lesz a paraméter a függvény hívásakor.

A másik megoldásunkban törekedtünk a használt programozási nyelv speciális eszközkészletének mind tökéletesebb kihasználására. Míg az előző kódot Pythonban nem, de más nyelven programozni tudó ember is hamar el tudja olvasni, meg tudja érteni, át tudja írni az általa használt nyelvre, a most következővel lényegesen nehezebben boldogulna. Az újabb megoldás előnye, hogy a nyelvhez értők sokkal könnyebben olvassák, módosítják, mert lényegesen letisztultabb. Minden modern programozási nyelvben megtaláljuk a csak az adott nyelvre jellemző sajátos megoldásokat, utasításokat.

A kód nagy része ez a jegyzék.

```
 1. def távirattá_alakít(szöveg):
 2.     helyettesítések = {'Á': 'AA', 'É': 'EE', 'Í': 'II',
 3.                        'Ó': 'OO', 'Ú': 'UU',
 4.                        'Ö': 'OE', 'Ő': 'OOE',
 5.                        'Ü': 'UE', 'Ű': 'UUE',
 6.                        '.': ' STOP', '?': ' STOP', '!': ' STOP',
 7.                        ',': '', ';': '', ':': '', '"': ''}
 8.     for helyettesítendő, helyettesítő in helyettesítések.items():
 9.         szöveg = szöveg.replace(helyettesítendő, helyettesítő)
10.     return szöveg
11.
12. print('Íme:', távirattá_alakít(input('Mi lesz a szöveg? ').upper()))
```

Kiválogatás helyett másolunk.

A cseréket olyan ciklus végzi, amelynek magja egyetlen sor. A replace() tagfüggvény a háttérben a fenti kódunk ciklusmagjához hasonló műveletsort végez. Csak nem látjuk.

Mindent bele! – Összefüggő feladatsor megoldása az eddigi nyelvi készletünkkel

Ahogy egy szép, harmatos reggelen szertenézünk nagy tölgyek alatt megbúvó, fehérre meszelt tanyánkon, kérdések és feladatok özöne kezd bennünket nyugtalanítani:

1. Hány állatunk van összesen?
2. Melyik állatunk a legöregebb?
3. Van-e olyan fajú állatunk, amelyet a felénk járó postás (a felhasználó) kérdezett?
4. Írjuk ki az állatfajok listáját! Kérjünk be egy fajnevet a felhasználótól! Írjuk ki, hogy az egyéves állataink között van-e ilyen fajú!
5. Melyik állatfajhoz tartozó állatból van a legtöbb?
6. Mennyi az állataink átlagéletkora fajonként?

És még egy feladatunk van:

7. Írjuk ki az egyes állatok neveit a fajuknak megfelelő nevű szövegfájlba!

Rendelkezésünkre áll a már ismert `allatok.txt` fájl.
Írjuk meg a fenti feladatokat megoldó programot `mindent_bele.py` néven!
Az első dolgunk egészen biztosan a fájl beolvasása lesz. Az adatokat ezúttal állatonként egy-egy szótárban helyezzük el, a szótár kulcsai: `név`, `faj` és `kor`. Az első két kulcs értéke karakterlánc, az utolsóé egész szám. A szótárak egy `állatok` nevű listába kerülnek.

```
állatok = []
with open('allatok.txt') as forrásfájl:
    for sor in forrásfájl:
        sor = sor.strip().split()
        állat = {'név': sor[0], 'faj': sor[1], 'kor':int(sor[2])}
        állatok.append(állat)
```

az üres lista

típusátalakítás

A szótárat a lista végéhez fűzzük.

egy állatot tartalmazó szótár

Az **1. feladat** megoldása a szótár elemszámának meghatározása. A feladat megoldása egyetlen sor, mi mégis két sort közlünk. A második sor ugyanazt eredményezi, mint az első – azaz a feladatsor megoldásához bőven elég az egyik. Ebben a leckében azonban futólag megismerkedünk a Python legújabb módszerével a karakterláncok formázására, és f-string formában is megadjuk néhány feladat megoldását. Az f-stringek onnan ismerhetők fel, hogy *f*-fel kezdődnek. Talán legfontosabb tulajdonságuk, hogy a string *belsejében* is szerepelhetnek változónevek vagy kiszámolandó kifejezések, mégpedig {kapcsos zárójelben}.

```
print(len(állatok), 'állatunk van összesen.')
print(f'{len(állatok)} 'állatunk van összesen.')
```

hagyományos megoldás

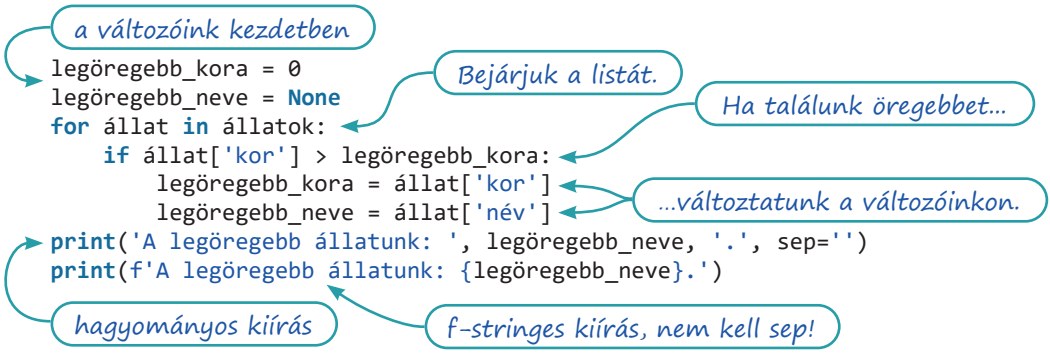
két rész, vesszővel felsorolva

egyetlen string

az f-string kezdő f-je

kiszámolandó kifejezés {}-ben

A **2. feladat** egy maximumkiválasztás.



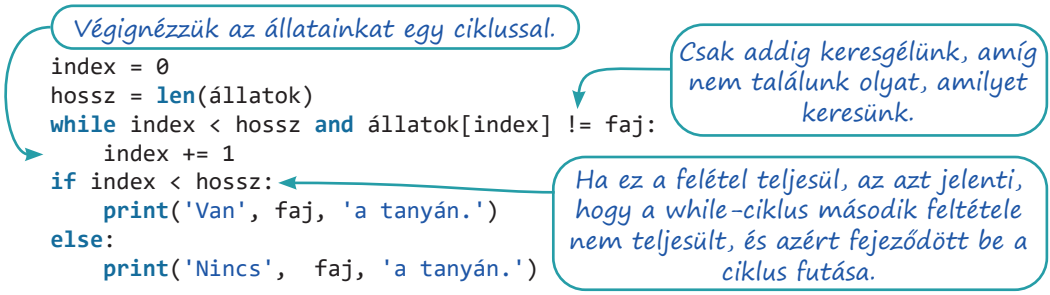
Minden magyarázat nélkül csak megjegyezzük, hogy a fenti sok sor helyett van jóval pythonosabb, a fentivel egyező eredményt adó megoldás is:

```
legöregebb_neve = max(állatok, key=lambda x: x['kor'])['név']
print(f'A legöregebb állatunk: {legöregebb_neve}.')
```

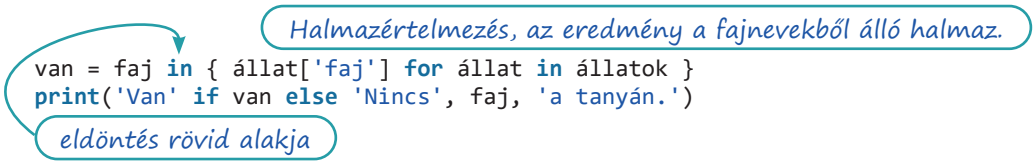
A **3. feladatban** bekérjük a fajt a postástól:

```
faj = input('Adjon meg egy fajnevet! ')
```

Innentől a feladat egy eldöntés.

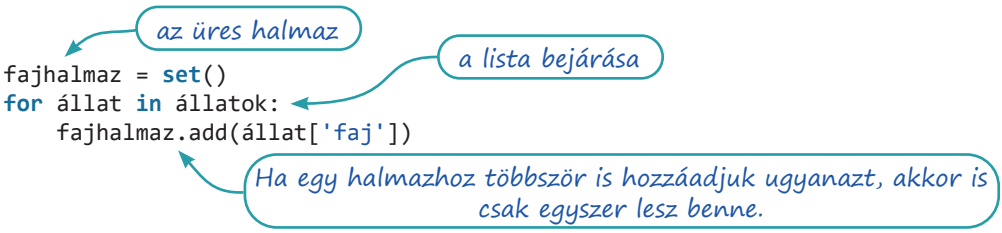


Egy, a fentivel azonos eredményt adó, igazán pythonos megoldás:



Természetesen a másodikként ismertetett megoldás kevésbé hatékony, mivel nem lépünk ki `break`-kel a halmaz előállításából, ha már találtunk a keresett fajú állatból. Ez a különbség csak nagyon-nagyon sok elemű adatszerkezetek átnézésekor lesz jelentős.

A **4. feladat** első részében már muszáj előállítanunk azt a halmazt, amelyet az imént csak ínyenceknek szóló megoldásmintánkban láttunk. Mindenképp halmazt érdemes előállítanunk, hiszen egy listában többször is előfordulna ugyanaz a faj. Ezúttal a hagyományos módon felírt ciklussal fogunk a feladathoz.




```
fajhalmaz = set()
for állat in állatok:
    fajhalmaz.add(állat['faj'])
```

Kiírjuk a halmaz elemeit (a tanyánkon előforduló fajokat), majd bekérjük a felhasználótól, hogy melyik érdekli.

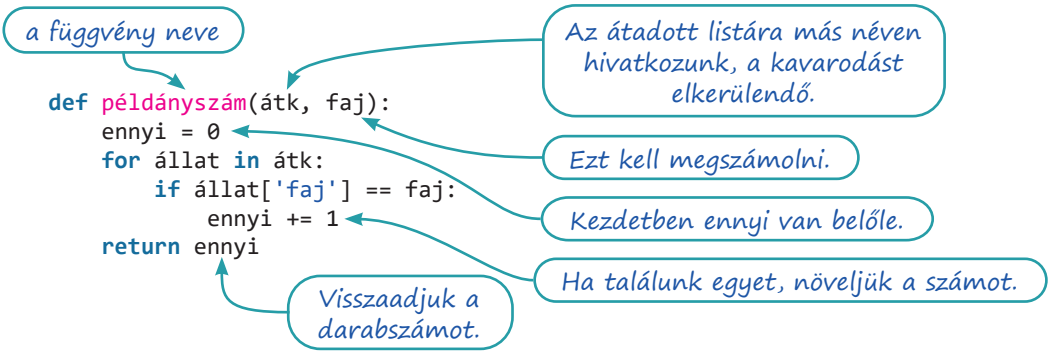

```
print('Állatfajok listája: ', ', '.join(fajhalmaz), '.', sep='')
print(f'Állatfajok listája: {", ".join(fajhalmaz)}.')
faj = input('Válasszon egy fajnevet! ')
```

Az f-string használatán alapuló megoldásban azért kell idézőjel az aposztróf helyett, mert az aposztróf lezárná magát a stringet, és ezt mindenképp el akarjuk kerülni.

Ezt követően már csak egy, az előző feladathoz hasonló eldöntés következik. A különbség az, hogy ezúttal két feltételünk is van: az állat faja és a kora.

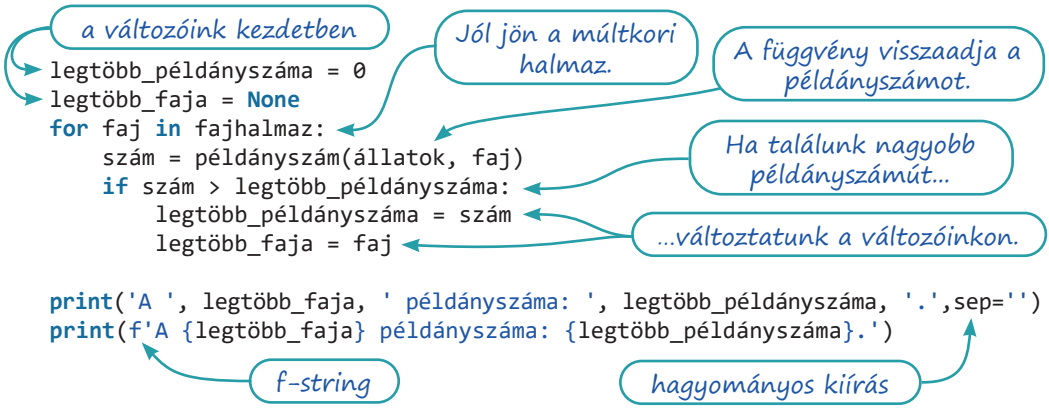
```
van = False
for állat in állatok:
    if állat['faj'] == faj and állat['kor'] == 1:
        van = True
print('Van' if van else 'Nincs', 'egyéves', faj, 'a tanyán.')
```

Az **5. feladatban** eljön az ideje egy függvény megírásának, hiszen egy olyan számlistának kell kiválasztanunk a maximumát, amelyet előbb létre kell hoznunk. A `példányszám` függvény megszámolja, hogy a megadott fajnak hány képviselője él a tanyán. Két paramétert vár: az `állatok` listát és egy fajnevet.



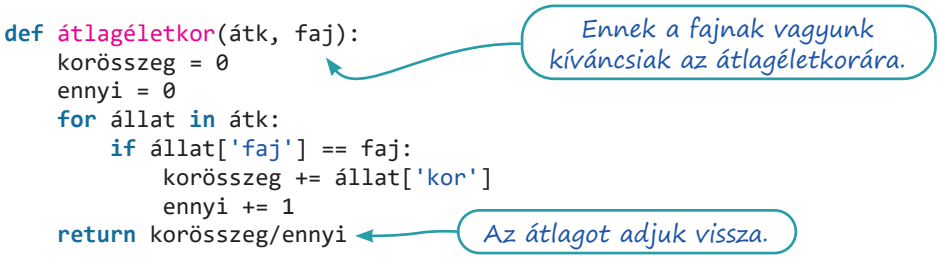

```
def példányszám(átk, faj):
    ennyi = 0
    for állat in átk:
        if állat['faj'] == faj:
            ennyi += 1
    return ennyi
```

A fenti függvényt a főprogramban hívjuk. Ez a programrész egy maximumkiválasztás, de a maximális érték megkeresésével párhuzamosan megkeressük a hozzá tartozó állatfajt is – ezzel megspórolunk egy külön keresést.



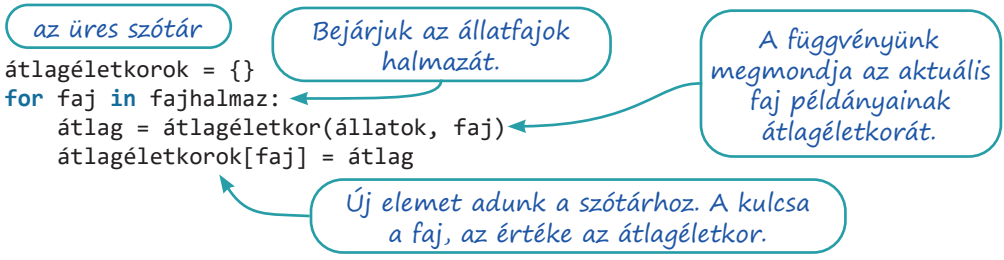

```
legtöbb_példányszáma = 0
legtöbb_faja = None
for faj in fajhalmaz:
    szám = példányszám(állatok, faj)
    if szám > legtöbb_példányszáma:
        legtöbb_példányszáma = szám
        legtöbb_faja = faj

print('A ', legtöbb_faja, ' példányszáma: ', legtöbb_példányszáma, '.',sep='')
print(f'A {legtöbb_faja} példányszáma: {legtöbb_példányszáma}.')
```

A **6. feladat** megoldását is egy függvény megadásával kezdjük. A függvény a paraméterként átadott faj átlagéletkorát adja vissza.




```
def átlagéletkor(átk, faj):
    korösszeg = 0
    ennyi = 0
    for állat in átk:
        if állat['faj'] == faj:
            korösszeg += állat['kor']
            ennyi += 1
    return korösszeg/ennyi
```

A fenti függvény hívásával egy szótárat állítunk elő a főprogramban. A szótár kulcsai az állatfajok, az értékek az átlagéletkorok.




```
átlagéletkorok = {}
for faj in fajhalmaz:
    átlag = átlagéletkor(állatok, faj)
    átlagéletkorok[faj] = átlag
```

Ezúttal külön-külön mutatjuk be a hagyományos kiírást és az f-string használatával működőt. Természetesen mindkettő az új szótárunk kulcs-érték párjainak bejárásán alapul.


```
for faj, átlagéletkor in átlagéletkorok.items():
    print(faj, ': ', átlagéletkor, sep='')
```

Az f-string lehetőséget nyújt a kiírt adatok balra, jobbra és középre rendezésére. A rendezések közül a <, a > és a ^ karakterrel adhatjuk meg a nekünk tetszőt. Megmondhatjuk azt is, hogy hány karakter széles lehet a rendezéssel kialakuló oszlop, és számok esetén a megjelenítendő tizedesjegyek számát is. Ezt használjuk most ki.

A fajnevet 14 karakteren, jobbra rendezve (figyeljük a csibecsőrt) írjuk ki.

```
for faj, átlagéletkor in átlagéletkorok.items():
    print(f'{faj:>14}: {átlagéletkor:>4.2f}')
```

Az átlagéletkort 4 karakteren, 2 tizedesre kerekítve, szintén jobbra rendezve írjuk ki.

Ehhez hasonló eredményt kell kapnunk:

```
        birka:  3.50
       malac:  1.67
        liba:  2.00
       kakas:  1.00
       kutya:  8.00
      kecske:  5.00
      macska:  2.50
  szarvasmarha:  3.00
       kacsa:  1.50
```

Kerekített érték – hasonlítsuk össze a régimódi kiírásnál látottal!

A **7. feladatban** két, egymásba ágyazott ciklusunk lesz. A külső végiglépked a fajok halmazának tagjain, a belső pedig az `állatok` listában keresi az épp aktuális fajhoz tartozó egyedeket.

Bejárjuk az állatfajok halmazát.

Megnyitjuk a fajnévnek megfelelő fájlt.

```
for faj in fajhalmaz:
    with open(faj + '.txt', 'w') as célfájl:
        for állat in állatok:
            if állat['faj'] == faj:
                print(állat['név'], file=célfájl)
```

Végigjárjuk az „állatok” listát.

Ha a fajba tartozó állatot találunk...

...a nevét beleírjuk a fájlba.

Megválaszoltuk a kérdéseket, felszáradt a harmat is, kezdődhet a nap!

Rend a lelkünk – A rendezés típusalgoritmusa és rendezés a napi gyakorlatban

Adatokat számítógéppel rendezni mind a mai napig az informatika egyik legérdekesebb problémája. Mindennapos élményünk, hogy egy weboldal – kereső, webshop, internetes adatbázis – sok ezer találatot, terméket, szócikket jelenít meg egyik-másik szempont szerint rendezve. A rendezéseknek ilyenkor nagyon gyorsan kell megtörténniük, különben a weboldal látogatója elunja a várakozást, és továbbáll. Az alkalmazások tervezői praktikák sokaságát vetik be, hogy a rendezett eredmények minél gyorsabban megjelenjenek, de a gyors eredmény alapja mindig a megfelelően megválasztott, hatékony rendezési típusalgoritmus.

Ha megijednénk, hogy ezek szerint rendezésből több is van, akkor megnyugtatásul elmondjuk, hogy bár az ijedelem megalapozott, találunk majd megnyugtató megoldást.

Egyszerű cserés rendezés

Sokféle rendezési típusalgoritmus létezik, amelyek nemcsak működésükben különböznek, hanem abban is, hogy milyen adathalmazt rendeznek gyorsan, hatékonyan. A hatékonyabb algoritmusok általában bonyolultabbak is, de mi most csak a legegyszerűbbel ismerkedünk meg. Számunkra ez már csak azért is elegendő, mert a mi adataink elemszáma elég kicsi, így nincs égető szükségünk hatékony rendezésre.

Az egyszerű cserés rendezés lényege, hogy az adatsor első elemét összehasonlítjuk az összes mögötte lévővel, és amelyik a későbbiek közül kisebb, azzal kicseréljük. Így a nagyobb elem az adatsor vége felé mozdul, a kisebb pedig az eleje felé. Ugyanezt a műveletet megismételjük az adatsor második, harmadik és az összes többi elemével, így a végén rendezett adatsort kapunk.

Futtassuk az alábbi programot, és nézzük meg a lista alakulását!

```
1. lista = [5, 3, 9, 1, 7]
2.
3. for egyik in range(len(lista)-1):
4.     for másik in range(egyik+1, len(lista)):
5.         print(lista, 'ezeket hasonlítom:', egyik, másik, end=' ')
6.         if lista[egyik] > lista [másik]:
7.             lista[egyik], lista[másik] = lista[másik], lista[egyik]
8.             print('csere volt, új sorrend:', lista[egyik], lista[másik],
9.                   'a lista a csere után:', lista)
10.        else:
11.            print('nem cserélünk')
```

Mindegyik elemet összehasonlítjuk...

...az összes mögötte lévővel.

Ha kell, cserélünk.

A belső ciklus mindig a *soron következő indexű* helyre keresi meg az oda való elemet. Amikor először fut végig, a 0. indexű helyre kerül a legkisebb elem. Amikor másodszor fut végig, az 1. indexű helyre kerül a második legkisebb elem. És így tovább.

Ezt és sok más algoritmust is *eltáncol* a Maros Művészegyüttes a Sapientia Erdélyi Magyar Tudományegyetem munkatársai rendezésében készült videókon, ezenkívül további érdekes animációkat is érdemes megnézni az interneten (például http://anim.ide.sk/rendezesi_algoritmusok_1.php).

A sorted() függvény és a lista.sort() tagfüggvény

Most, hogy van elképzelésünk a rendezés működéséről, lássuk a megnyugtató megoldást! A Pythonnak és a modern programozási nyelveknek általában van beépített parancsuk, függvényük a rendezésre. A Pythonnak rögtön kettő is. Nézzük meg a működésüket, futtassuk az alábbi programot!

```
1. lista = [5, 3, 9, 1, 7]
2.
3. rendezett = sorted(lista)
4. print(lista, rendezett)
5.
6. fordítva_rendezett = sorted(lista, reverse=True)
7. print(lista, fordítva_rendezett)
8.
9. lista.sort()
10. print(lista)
11.
12. lista.sort(reverse=True)
13. print(lista)
```

Az első paranccsal az eredeti lista nem változik, a rendezés eredményét nekünk kell kiírni vagy új listába tenni.

Fordított irányba is tudunk rendezni.

A második módszer az eredeti listában rendezi sorba az elemeket. Úgy mondjuk, hogy helyben rendez. Az eredeti sorrend elvész.

Ezzel is tudunk fordított irányba rendezni.

Természetesen a rendezés szövegek esetében is működik. A rendezés alapját alapesetben a karakterek kódolás szerinti sorrendje képezi – arról a kódolásról van szó, amelyről a fájlbeolvasáskor már szót ejtettünk. A lényeg – ha csak a latin ábécén alapuló nyelveket tekintjük – az, hogy az ékezetes betűt nem tartalmazó karakterláncok minden további nélkül rendezhetők. Az angol ábécé betűi ugyanis az ábécésorrendnek megfelelő sorrendben kaptak számkódokat. Az „A” kódja kisebb, mint a „B” kódja, és így tovább. A kisbetűk a nagybetűk után következnek, azaz a „Z” előbb következik, mint az „a”. Ha a rendezendő szavak eleje egyezik, a `sorted()` függvény ügyesen megnézi a többi karaktert is. Próbáljuk ki!

```
1. lista = ['Dalmata', 'bakancs', 'alma', 'alap']
2. print(sorted(lista))
```

Az egyes nyelvek ábécéjének figyelembevétele rendezéskor már egészen más szintű probléma. A megoldáshoz egyes programozási nyelvekben ügyeskednünk kell egy keveset, másokban pedig jó sokat – de ezt a problémakört meghagyjuk azoknak, akik a középiskola elvégzését követően is szívesen programoznának.

27. példa: A tanyán élő állataink neve és kora

Újabb példánkban tanyánk állatai közül a kutyák és a macskák szerepelnek, no meg a kakasok – rájuk mindig számíthatunk, ha nem akaródzik hajnalban kelni. Írjuk ki a nevüket a `rendezett_allatok.py` programban előbb ábécérendben, majd kor szerinti sorrendben, a legöregebbtől a legfiatalabbig! Utóbbi esetben kihasználhatjuk, hogy nincs két azonos korú kedvencünk.

Gondoljuk át, miként oldjuk meg ezt a feladatot!

- Beolvassuk a fájl adatait, és tároljuk őket egy kétdimenziós listában – ezt már többször megoldottuk, csak le kell másolnunk valamelyik régebbi programunk elejét.
- Első részfeladat:
  - Gyűjtsük listába a kutyák, a macskák és a kakasok nevét!
  - Rendezzük a listát!
  - Írjuk ki a rendezett névlista elemeit!
- Második részfeladat:
  - Gyűjtsük listába a három faj egyedeit (nemcsak a nevet, hanem az egész kis listát, mert a korra is szükség lesz)!
  - Gyűjtsük ki a korokat!
  - Rendezzük a korokat fordított sorrendben!
  - Járjuk be a korokat, keressük meg és írjuk ki az egyes korokhoz tartozó állatokat!

(Ha szívesebben dolgozunk listákat tartalmazó listákkal, akkor megtehetjük, hogy nemcsak az állatok nevét gyűjtjük ki az első lépésben, hanem az egyes állatok kis listáit is, és így oldjuk meg a feladatot.)

Kódoljuk a feladat megoldását!

```
 1. állatok = []
 2.
 3. with open('allatok.txt') as forrásfájl:
 4.     for sor in forrásfájl:
 5.         sor = sor.strip().split()
 6.         sor[-1] = int(sor[-1])
 7.         állatok.append(sor)
 8.
 9. nevek = []
10. for állat in állatok:
11.     if állat[1] in ['kutya', 'macska', 'kakas']:
12.         nevek.append(állat[0])
13.
14. print('Kedvenceink ábécérendben:\n', '\n'.join(sorted(nevek)), sep='')
15.
16. korok = []
17. kedvencek = []
18. for állat in állatok:
19.     if állat[1] in ['kutya', 'macska', 'kakas']:
20.         korok.append(állat[-1])
21.         kedvencek.append(állat)
22.
23. korok.sort(reverse=True)
24.
25. print('Kedvenceink kor szerint:')
26. for kor in korok:
27.     for kedvenc in kedvencek:
28.         if kor == kedvenc[-1]:
29.             print(kor, kedvenc[0])
30.
```

Tároljuk a fájlban lévő adatokat.

Kigyűjtjük a neveket.

rendezés

Az ábécésorrendben lévő neveket sortöréssel elválasztva írjuk ki.

Kigyűjtjük a korokat.

Kigyűjtjük a kedvenc állatainkat.

Rendezzük a korokat.

Megkeressük és kiírjuk az egyes korokhoz tartozó állatneveket.

Megjegyezzük, hogy a Python `sort()` és `sorted()` függvénye egyébiránt remek dolgokat tud, például a *kedvencek* kétdimenziós listát a kis listák utolsó eleme – a kor – alapján akár fordított irányba is képes rendezni. A szintaxis nem egyszerű, de az érdeklődők számára megmutatjuk:

```
kedvencek.sort(key=lambda x: x[-1], reverse=True)
```

Eggyé válás – A metszetképzés és az egyesítés (unió) típusalgoritmusa

Típusalgoritmusokkal foglalkozó leckéink közül az utolsóban két olyan módszerrel ismerkedünk meg, amelyek nem is egy, hanem két vagy több kiindulási adatszerkezethez rendelnek egyetlen újat.

A metszetképzés és az unió valójában egyaránt halmazművelet. Két halmaz metszete egy olyan halmaz, amelyik a két halmaz közös elemeit tartalmazza. Két halmaz uniója egy olyan halmaz, amelyik a két halmaz minden elemét tartalmazza – a közöseket is, meg azokat is, amelyek csak az egyik kiindulási halmazban vannak benne.

Vannak olyan programozási nyelvek, amelyekben nincs halmaz adattípus. Az ilyenekben a halmazokat valamilyen egyéb bejárható adatszerkezettel – például listával, tömbbel – valósítjuk meg. Metszetképzéskor a két lista közös elemeit írjuk újabb listába, egyesítéskor pedig az egyik lista elemeit bővítjük a második listában lévő, az elsőben eddig nem szereplő elemekkel. A listák és a tömbök azonban két fontos dologban különböznek a halmazoktól:

- a listákban és a tömbökben előfordulhat azonos elem többször is, a halmazokban nem;
- a listákban és a tömbökben kötött sorrendjük van az elemeknek, a halmazokban nincs.

Metszetképzés

28. példa: Tömegközlekedés Százszorszépvárosban

Százszorszépvárosban szeretnénk közösségi közlekedéssel eljutni a 92-es villamos végállomásáról, a Fapapucs sugárútról a 19-es busz Balambér tér nevű megállójába. A két járat megállói listaként állnak rendelkezésünkre. Melyik megállókban tudunk átszállni a villamosról a buszra?

A közös megállók nevére van szükségünk, azaz metszetet kell képeznünk.

A következő műveleteket kell kódolnunk az `atszallas.py` nevű programban:

- Felveszünk egy üres listát, például `átszállások` néven.
- Ciklussal bejárjuk a villamos valamennyi megállóját.
- Ha az aktuális villamosmegálló a buszjáratnak is megállója, és még nem szerepel az `átszállások` listában, akkor hozzáfűzzük.

Kódoljuk a programot!

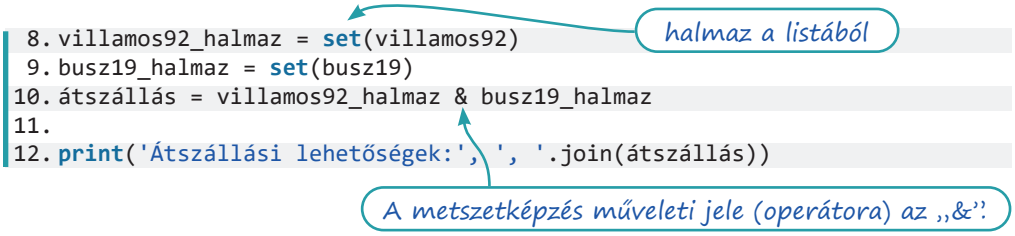
```
villamos92 = ['Kiss u.', 'Zöld fasor', 'Piros tér',
              'Malom-patak', 'Puccos u.',
              'Remegő-erdő', 'Fapapucs sgt.']
busz19 = ['Nagy u.', 'Balambér tér', 'Szent Lajos hídja',
          'Által-ér', 'Reghős Bendegúz sz.k.', 'Puccos u.',
          'Remegő-erdő', 'Pöszke-liget']

átszállás = []
for villamosmegálló in villamos92:
    for buszmegálló in busz19:
        if villamosmegálló == buszmegálló and \
           villamosmegálló not in átszállás:
           átszállás.append(villamosmegálló)
```

Az if villamosmegálló in busz19 kódrészlet felhasználásával egyszerűsíthetjük az algoritmust.

Vigyázzunk, hogy ne legyen ismétlődő elem!

```
14.             break
15.
16. print('Átszállási lehetőségek:', ', '.join(átszállás))
```

Ha a használt programozási nyelvnek van halmaz adattípusa, a dolgunk sokat egyszerűsödik. A kódot csak a 8. sortól közöljük, addig ugyanis nincs változás:




```
 8. villamos92_halmaz = set(villamos92)
 9. busz19_halmaz = set(busz19)
10. átszállás = villamos92_halmaz & busz19_halmaz
11.
12. print('Átszállási lehetőségek:', ', '.join(átszállás))
```

Említettük már, hogy a halmazokban nem definiált az elemek sorrendje. Futtassuk le a programunkat néhányszor, és látni fogjuk, hogy a két átszállási kapcsolatot nem minden alkalommal írja ki ugyanabban a sorrendben.

Említettük azt is, hogy a halmazokba nem kerülhet kétszer ugyanaz az elem. Ezt arra is kihasználhatjuk, hogy az ismétlődéseket tartalmazó listából eltüntessük az ismétlődéseket úgy, hogy a listát előbb halmazzá alakítjuk, majd a halmazt újra listává:

```
új_lista = list(set(ismétlődést_tartalmazó_lista))
```

A módszer nem használható, ha fontos az eredeti lista elemeinek sorrendje.

Unió

29. példa: Fricska és Kökény első szavainak jegyzéke

A szomszédban lakó két bölcsis – Nagy Fricska és Zsémbes Kökény – első hét, illetve nyolc szavát feljegyezték anyukáik. Közösen tartott névnapi zsúrjukon olyan dalt akarnak nekik énekelni, amely emléket állít első szavaiknak. Állítsuk össze azt a szójegyzéket, amelyben a két tipegő minden feljegyzett szava szerepel, ismétlődés nélkül!

A két jegyzék minden szavára szükségünk van, tehát az egyesítés típusalgoritmusát használjuk.

A következő műveleteket kell kódolnunk, ha nem használunk halmaz adattípust:

- Felveszünk egy üres listát.
- Az üres listába átmásoljuk az első lista elemeit.
- Az első lista elemeit tartalmazó listába átmásoljuk a második lista elemeit, figyelve, hogy csak azt másoljuk, ami még nincs benne.

Végezzük el a kódolást a `kozos_szavak.py` programban!

Mielőtt nekifogunk a kódolásnak, kitérünk a Python egyik, a listák másolásával kapcsolatos sajátságára. Ha csak annyit írunk a kódba, hogy

```
másolat = kiindulási_lista
```

akkor a másolat csak egy újabb hivatkozás lesz a számítógép memóriájában lévő eredeti listánkra. Ha ilyenkor a kiindulási listán változtatunk – mondjuk újabb elemet fűzünk

a végére, vagy kicserélünk egy már meglévő elemet –, akkor a változást a másolat is átveszi. A problémán úgy lehetünk úrrá például, hogy a fenti sor helyett a

```
másolat = kiindulási_lista[:]
```

parancsot adjuk ki.

```
 1. fricska = ['anya', 'maci', 'baba',
 2.            'apa', 'kaja', 'ló',
 3.            'erősáramú feszültségszabályzó']
 4. kökény = ['apa', 'autó', 'anya',
 5.           'víz', 'maci', 'könyv', 'nagyi',
 6.           'pörgettyűs tájoló']
 7.
 8. szójegyzék = fricska[:]
 9. for szó in kökény:
10.     if szó not in szójegyzék:
11.         szójegyzék.append(szó)
12.
13. print('Fricska első szavai:', ', '.join(sorted(fricska)))
14. print('Kökény első szavai:', ', '.join(sorted(kökény)))
15. print('A kész szójegyzék:', ', '.join(sorted(szójegyzék)))
```

Az első lista elemeit másoljuk az újba.

Új elemekként felvesszük Kökénynek azokat a szavait, amiket Fricska nem mondott.

Csak azért rendezünk, hogy könnyebben lássuk az eredményt.

Ha a használt programozási nyelv ismeri a halmaz adattípust, a dolgunk ezúttal is egyszerűsödik. A kódot csak a 8. sortól közöljük, addig megegyezik az előzővel.

```
 8. fricska_halmaz = set(fricska)
 9. kökény_halmaz = set(kökény)
10.
11. szójegyzék = fricska_halmaz | kökény_halmaz
12.
13. print('Fricska első szavai:', ', '.join(sorted(fricska_halmaz)))
14. print('Kökény első szavai:', ', '.join(sorted(kökény_halmaz)))
15. print('A kész szójegyzék:', ', '.join(sorted(szójegyzék)))
```

Az egyesítés műveleti jele (operátora) a „|” („cső”, Alt Gr+W magyar billentyűzetkiosztásnál).

Csoportnapló – Tapogatózás az objektumorientált programozás irányába

Ebben a leckében egyetlen program elkészítése a feladatunk. A feladatot az eddig tanult eszközeinkkel fogjuk megoldani, több alkalommal meg-megállva. Egy-egy ilyen megtorpanást arra használunk fel, hogy arról elmélkedjünk, milyen problémákkal szembesülünk, mennyire jó a megoldásunk, és hogy mi segíthetne jobbá tenni. Ha eltekintünk az elmélkedésektől, felfoghatjuk egyszerű gyakorlásnak is.

30. példa: Csoportnapló
Adott egy fájl az alábbi vagy hozzá hasonló tartalommal:

Mint azt látjuk, a fájl lehetne akár egy vegyes (több osztály diákjait is magába foglaló) tanulócsoport, például egy nyelvórai csoport tanárának feljegyzése a csoportba járó diákok jegyeiről.

A fájl beolvasását követően a következő feladatokat kell megoldanunk a `csoportnaplo.py` programban:

1. Írjuk ki a 11. zs osztályos diákok nevét!
2. Írjuk ki azoknak a nevét, akik nem írták meg mindhárom idei témazárót, és soroljuk fel a megírt témazáróiknak a jegyeit!
3. Határozzuk meg, ki írta a 11. x osztályban a legkevesebb témazárót!
4. Az utolsó előtti órán felelésre számítanak a diákok. A diákok szerint, amikor felelés van, az szokott felelni, akinek kevés a jegye. A tanárnő ennél pontosabb: azok közül válogat felelőt, akiknek a rendes jegyeinek száma kevesebb, mint a legtöbb rendes jeggyel bíró diák rendes jegyeinek 80 százaléka. Írjuk ki azoknak a nevét, akik „veszélyeztetettek”!
5. A tanárnő mégsem feleltet, hanem lezárja a diákok év végi jegyeit. Határozzuk meg és írjuk ki a diákok átlagát és év végi jegyét! Az átlagszámításkor a témazárók duplán számítanak. Az év végi jegy 1,7-es átlag fölött kettes, 2,5 fölött hármas, 3,5 fölött négyes, és 4,5-től ötös.

Gondoljuk végig, hogy az egyes kérdésekhez melyik típusalgoritmus szükséges!

1. kiválogatás
2. kiválogatások (hol van *t* betű a szám után) alapján megszámlálások (a témazárók száma); az eredmény alapján kiválogatás (nevek), végül újabb kiválogatások (a jegyek kiírása – elképzelhető, hogy az első kiválogatások eredményét használjuk majd)

3. minimumkiválasztás
4. megszámlálások, maximumkiválasztás, majd kiválogatás
5. sorozatszámítások

Mint ahogy már többször tudatosult bennünk, a jó adatszerkezet megkönnyíti a jó program írását. Határozzunk az adatok tárolásának módjáról!

Sok diák adatait kell tárolnunk. Egy-egy diáknak három *tulajdonságát* figyeljük meg:

- a nevét,
- az osztályát és
- a jegyeit.

A jegyek tűnnek a legmacerásabbnak, és elég sok művelet van velük kapcsolatosan. A forrásfájlban a témazárókra kapott jegyek egy listát alkotnak a rendes (nem témazáróra kapott) jegyekkel, alighanem azt a sorrendet tükrözve, ahogy a tanár feljegyezte őket. A kérdések között semmi nem vonatkozik a jegyek sorrendjére, az viszont, hogy témazáróra kapta-e a diák a jegyet, vagy sem, több esetben is fontos lesz. A jegyek elkülönítését az előzőek fényében érdemesnek tűnik már a fájl beolvasásakor megtenni. Érdemes-e már a beolvasáskor átalakítani a jegyeket számmá? Számolni fogunk velük, tehát érdemes.

Az osztályokkal kapcsolatos tevékenységeinket vizsgálva észrevesszük, hogy mindösszesze egyetlen feladatnál érdekes az évfolyam, azaz talán nem szükséges külön tárolnunk az évfolyamot és a betűjelet.

A neveket elég csak kiírni, és soha nem kell külön kezelnünk a vezeték- vagy keresztneveket. A nevek tárolhatók egyetlen karakterláncként.

Az eddigiek alapján érdemes lesz diákonként egy-egy szótárat létrehozni. Hangsúlyozzuk, hogy a feladat megoldható összetett listával is, de így tudunk az egyes *tulajdonságokra* név szerint hivatkozni.

- A szótár elsőként létrehozott kulcsa: „név”, az érték a diák neve.
- A második kulcs: „osztály”, az érték a diák osztálya olyan formában, ahogy a fájlban is szerepel.
- A harmadik kulcs: „rendes_jegyek”, az érték a rendes jegyeket egész számként tartalmazó lista.
- A negyedik kulcs: „témazáró_jegyek”, az érték az előzőhöz hasonló.

A beolvasás folyamán létrejött szótárakat egy listában helyezzük el.

A megoldás

Írjuk is meg a beolvasást végző részt! A közölt kód a 6. sortól kezdődik, mert elé még majd kerül egy függvény. Most az alábbi formában futtatható:

```
 6. diákok = []
 7. with open('naplo.txt') as forrásfájl:
 8.     for sor in forrásfájl:
 9.         sor = sor.strip().split(', ')
10.         diák = {}
11.         diák['név'] = sor[0]
12.         diák['osztály'] = sor[1]
13.         jegyek = sor[2].split()
14.         diák['rendes_jegyek'] = []
```

üres lista a szótáraknak

Soronként beolvassuk a fájlt.

A vessző + szóközök mentén darabolva háromtagú listává alakítjuk a sort.

A jegyeket tartalmazó részt szétdaraboljuk a szóközök mentén.

```
14.         diák['rendes_jegyek'] = []
15.         diák['témazáró_jegyek'] = []
16.         for jegy in jegyek:
17.             if len(jegy) == 1:
18.                 diák['rendes_jegyek'].append(int(jegy))
19.             else:
20.                 diák['témazáró_jegyek'].append(int(jegy[0]))
21.         diákok.append(diák)
```

Ha nincs t, akkor egy karakter hosszú egy jegy, ha van t, akkor kettő.

A t betűt nem tesszük el.

Az **1. feladatot** megoldó programrészlet (a sorok számozásakor számoljunk egy-egy, a kód olvashatóságát megkönnyítő üres sorral):

```
23. print('11. évfolyamra jár:')
24. for diák in diákok:
25.     if diák['osztály'][:2] == '11':
26.         print(diák['név'])
```

Itt állítjuk elő az évfolyamot az osztály jeléből.

A kód ennyi programozásóra után aligha ígért jelentősebb fejtörést, de alkalmat ad az első töprengésünkre.

Megegyeztünk abban, hogy a mostani programunk nem teszi szükségessé az osztály, azaz az évfolyam és a betűjel együttesének bonyolultabb tárolását. Nagyon könnyen el tudunk azonban képzelni olyan programot – elég csak az elektronikus naplóra gondolnunk –, ahol sok-sok programrész foglalkozik ezzel az adattal. Hol csak az évfolyamra van szükség, hol csak a betűjelre, hol mind a kettőre, de néha `11x`, máskor pedig `11. x` vagy `11/x` formában.

Természetesen az említett sok-sok programrész elő tudja magának állítani a megfelelő formát, pont úgy, ahogy a fenti kódrészlet 25. sorában mi is megtettük. De ez sok-sok újabb kódsort jelent, ami pedig – mostanra saját tapasztalataink alapján tudjuk – sok-sok hibalehetőséget hordoz magában. Milyen remek volna, ha lenne olyan, hogy `diák['évfolyam']` és `diák['oszálybetű']`!

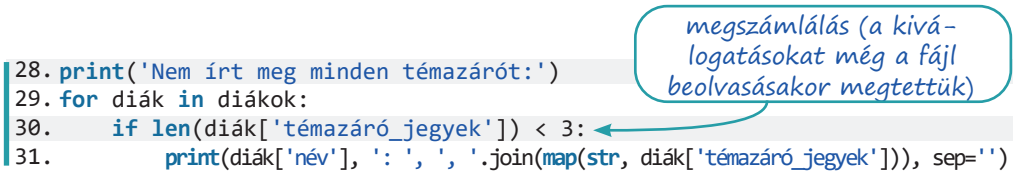
Ó, hiszen ilyet tudunk létrehozni! Egyszerűen újabb sorokat írunk a fájlbeolvasó részbe, és lesznek ilyen adataink!

Igen, létre tudunk hozni ilyen kódrészletet, de ne feledjük, hogy ilyenkor ugyanazt az adatot tároljuk többször. Ez legalább két okból problémás. Az egyik, hogy sok tárterületet foglal. A másik, hogy sok hibalehetőséget rejt magában. Ha például itt az év vége, és a `11x`-et `12x`-re kell javítani, akkor hány helyen is kell frissítenünk az adatokat? Előbb-utóbb valamelyikről elfelejtkezünk. Ez a megoldás tehát nem igazán jó megoldás.

Akkor írjunk rá függvényt! Lenne `évfolyam(diák)` függvényünk, átadnánk neki egy diákot (a diákot tartalmazó szótárt), a visszatérési érték pedig az évfolyam száma lenne.

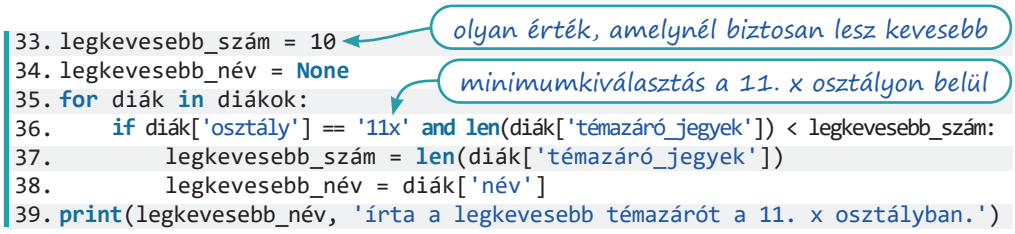
Ez a megoldás kiküszöböli az előző hiányosságait, cserébe viszont újakat teremt. Képzeljük el, hogy egy nagy programot fejlesztő programozói csoportba új ember érkezik. A kód hatalmas. Honnan fog rájönni az új programozó, hogy a diákok évfolyamát könnyen megkapja az `évfolyam()` függvény használatával? Honnan jövünk rá mi magunk, ha fél év után újra ezzel a kóddal kell foglalkoznunk? Még nem válaszolunk erre a kérdésre.

A **2. feladatot** a következő programrészlettel oldhatjuk meg:



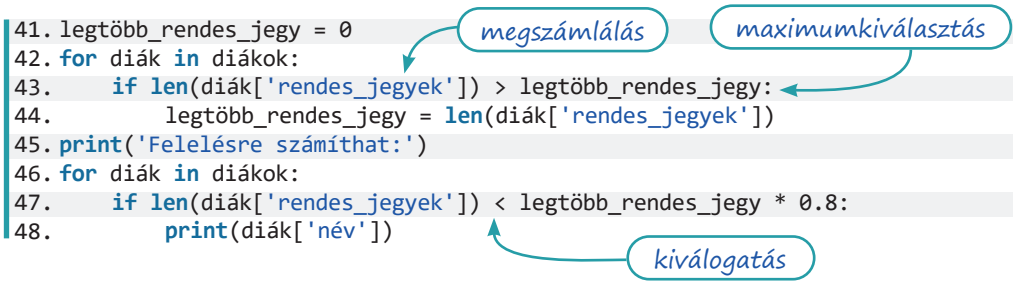

```
print('Nem írt meg minden témazárót:')
for diák in diákok:
    if len(diák['témazáró_jegyek']) < 3:
        print(diák['név'], ': ', ', '.join(map(str, diák['témazáró_jegyek'])), sep='')
```

A **3. feladatot** megoldó programrészlet:




```
legkevesebb_szám = 10
legkevesebb_név = None
for diák in diákok:
    if diák['osztály'] == '11x' and len(diák['témazáró_jegyek']) < legkevesebb_szám:
        legkevesebb_szám = len(diák['témazáró_jegyek'])
        legkevesebb_név = diák['név']
print(legkevesebb_név, 'írta a legkevesebb témazárót a 11. x osztályban.')
```

A **4. feladatot** a következő programrészlettel oldhatjuk meg:




```
legtöbb_rendes_jegy = 0
for diák in diákok:
    if len(diák['rendes_jegyek']) > legtöbb_rendes_jegy:
        legtöbb_rendes_jegy = len(diák['rendes_jegyek'])
print('Felelésre számíthat:')
for diák in diákok:
    if len(diák['rendes_jegyek']) < legtöbb_rendes_jegy * 0.8:
        print(diák['név'])
```

Ezen a ponton ismét megállunk töprengeni. Felidézzük azt a gondolatot, miszerint azért is fontos a könnyen olvasható program írása, mert egy kész programot manapság többször, sokszor olvas újra valaki azért, hogy megértse, mit csinál a program. Ez lehet az az érettségiző, aki reggel nyolckor programozással kezdte az érettségit, majd más feladatra váltott, és tizenegykor innen folytatná. De lehet az a fejlesztő is egy valós munkahelyen, akinek módosítania kell egy már működő és sokak által használt kódot, olyat, amit nem is ő írt.

Ha most ránézünk erre a kódra, látjuk-e szinte azonnal, hogy mit csinál? A 45. sor sokat segít, de bizonyosan további segítséget jelentene, ha a 47. sorban írhatnánk olyat, hogy `if diák.rendes_jegyek_száma()`. Szerencsére lesz erre lehetőség.

Az **5. feladat** megoldásához bontsuk a feladatot gondolatban két részre! Egyrészt meg kell határoznunk egy-egy diák átlagát, másrészt az átlag alapján megállapítani az év végi jegyét. Az átlag megállapítására függvényt írunk. Tudjuk, hogy a témazárók duplán számítanak, és szerencsére már a fájl beolvasásakor elkülönítettük a témazáróra kapott jegyeket a rendes jegyektől, most ezt kihasználva, külön listaként adjuk át a két jegylistát a függvényünknek. A függvény a következő formát ölti:

```
1. def átlag(rendes_jegyek, témazáró_jegyek):
2.     összeg = sum(rendes_jegyek) + sum(témazáró_jegyek) * 2
3.     osztó = len(rendes_jegyek) + len(témazáró_jegyek) * 2
4.     return round(összeg/osztó, 2)
```

Ahogy a sorszámozásból látszik, a kód elejére helyezzük el a függvényt – ami Pythonban nem kötelező, de általában nem bánjuk meg, ha így járunk el.

A kód végén a következőképpen hívjuk a függvényt:

```
50. for diák in diákok:
51.     á = átlag(diák['rendes_jegyek'], diák['témazáró_jegyek'])
52.     if á <= 1.7:
53.         jegy = 1
54.     elif 1.7 < á < 4.5:
55.         jegy = int(round(á, 0))
56.     else:
57.         jegy = 5
58.     print(diák['név'], 'átlaga:', á, 'jegye:', jegy)
```

Ezzel elkészültünk a feladat teljes megoldásával.

Utolsó töprengésünkben arra mutatunk rá, hogy ha nem mi írtuk a függvényt, akkor nem nyilvánvaló első pillantásra, hogy miért két paraméterrel kell hívnunk, és hogy mi ez a két paraméter. Mennyivel egyszerűbb volna azt írni a programunk 51. sorában, hogy `á = diák.átlag()`!

Gondoljuk egy kicsit tovább a problémát! Egy nagy programban – mint az említett e-napló – valószínűleg sok-sok helyen kell átlagot számolni. Tegyük fel, hogy egy napon az iskola elhatározza, hogy az órai munkára kapott jegyek az év végi átlag számításakor csak feleannyit érnek, mint egy „rendes” jegy. A fájl beolvasásakor hamar megoldható, hogy létrejöjjön egy `diák['órai_munka_jegyek']` lista, de ezt követően még *minden függvényhívást* módosítani kell, hiszen az `átlag()` függvénynek egy harmadik paramétert is át kell adnunk. Jó eséllyel valamelyiket elfelejtjük a sok közül, és akkor az e-napló egyik része szerint más lesz a diák átlaga, mint a másik rész szerint.

A töprengéseink során arra a megállapításra jutottunk, hogy

- elsősorban nagyobb programoknál,
- másodsorban a kód olvashatósága, azaz a kód átláthatósága és karbantarthatósága,
- végül a kód könnyebb bővíthetősége és a programozási folyamat egyszerűsödése miatt örülnénk, ha valamilyen, a fentieket megoldó eszközt kapnánk a kezünkbe. Szerencsére van ilyen eszköz, az úgynevezett objektumorientált programozás (röviden OOP), amely nem csak a fenti problémákra kínál megoldást.

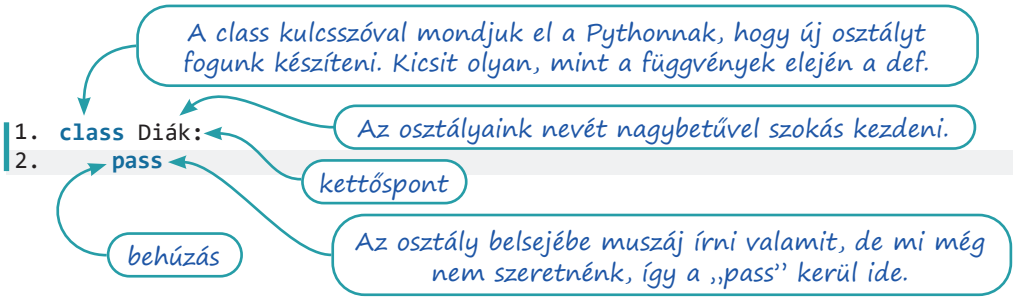
A módszer lényegének, fontosabb tulajdonságainak ismerete akkor is segítségünkre lehet, ha olyan rövid programokat írunk egymagunkban, mint amilyeneket eddig írtunk. Ennek az a magyarázata, hogy programjainkban gyakran támaszkodunk mások által megírt programrészekre. Tettünk már ilyet, a random modul használatával, de néhány leckével később grafikus felhasználói felülettel bíró programokat fejlesztünk majd, és ott nagyon hasznos, ha minél pontosabb elképzeléseink vannak az objektumorientált programok mibenlétéről.

A következő leckékben tehát az objektumorientált programozásról lesz szó.

A terv, amely nem megy füstbe – Saját adatszerkezetek tervezése és megvalósítása objektumosztályokkal

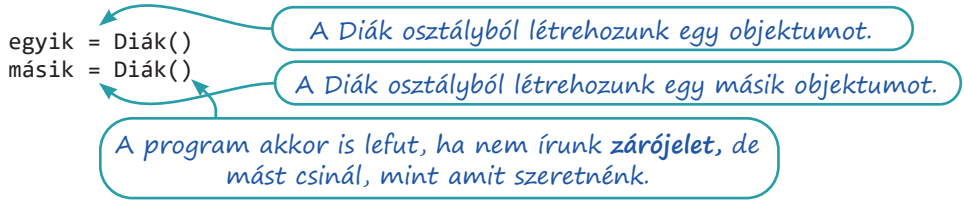
Eddig alapvetően háromféle összetett adattípussal – adatszerkezettel – dolgoztunk: listával, szótárral és halmazzal. Persze volt olyan listánk, amely listákat tartalmazott, és olyan is, amely szótárakat, illetve volt olyan szótárunk is, amelynek lista volt valamelyik értéke. Ebben a leckében megtanulunk a programunkhoz tökéletesen illeszkedő adatszerkezeteket létrehozni.

Egy ilyen, tökéletesen testreszabott adatszerkezet tervének neve: osztály, angolul: class. Lássunk egy ilyen tervet:



Az osztályokat szokás **objektumosztálynak** is nevezni, mert objektumokat hozunk belőlük létre. Az egy osztály alapján létrehozott objektumok elég hasonlóak, ezért azt mondjuk, hogy egy osztályba tartoznak.

Egészítsük ki a fenti kódot az alábbi két sorral! (A következő néhány példa sorait nem mindig számozzuk, mert csak próbálkozunk, gyakran fogunk törölni, új sorokat beírni.)



Van tehát két, `Diák` osztályú objektumunk. (Az, hogy ezek a diákok is iskolai osztályba járnak, puszta véletlen – ez egy másféle osztály.) Egy-egy objektumról azt mondjuk, hogy a tervéül szolgáló osztály egy példánya. Amikor létrehozzuk, akkor azt mondjuk, hogy **példányosítjuk** az osztályt.

A fenti példában a `Diák` tehát egy osztály, hosszabb nevén objektumosztály, azaz egy terv. Ennek a tervnek az alapján jön létre az `egyik` és a `másik` objektum, **objektumpéldány** vagy csak példány. Az utolsó két kódsor pedig két **példányosítás.** Az elsőben az `egyik`, a másodikban a `másik` nevű objektumot példányosítjuk a `Diák` osztályból.

De mi az az objektum?

Az objektumok egy-egy létező, azaz egy-egy személy, tárgy, fogalom számítógépes programban létrehozott reprezentációi, megvalósulásai. A létező lehet bármi: diák (mint a példánkban), kutya, egér, borsószem, érdemjegy, könyv, csavaranya, zeneszám, álom vagy éppen papucs orrán pamutbojt.

Az objektumokat ebben a leckében még csak arra használjuk, hogy a tárolt létezők bennünket érdeklő tulajdonságait tároljuk bennük. A tulajdonságok sokfélék lehetnek. A kutyának tárolhatjuk a nevét és a fajtáját, a borsószemnek a színét és az alakját, a könyvnek a szerzőjét, a címét és az oldalszámát, a zeneszámnak az előadóját és a hosszát. Az ilyen tulajdonságokat ebben a könyvben *jellemzőnek* nevezzük (szokás még többek között a mező, az adattag és a változó elnevezést használni).

Mi a diákoknak a vezeték- és a keresztnevét, valamint az évfolyamát tároljuk. Bővítsük a programunkat:

```
egyik.vezetéknév = 'Lopakodó'
egyik.keresztnév = 'Kasztanyetta'
egyik.évfolyam = 11
másik.vezetéknév = 'Surranó'
másik.keresztnév = 'Szalicil'
másik.évfolyam = 11
```

A tárolt jellemzőket ki is tudjuk írni:

```
print(egyik.keresztnév, másik.évfolyam)
```

A jellemzők értéke módosítható:

```
másik.évfolyam += 1
egyik.keresztnév = egyik.keresztnév + ' ' + 'Harmónia'
```

Az objektum inicializálása

Az objektumokat eddig úgy hoztuk létre, hogy példányosításukat követően az egyes jellemzőknek „kézzel” adtunk értéket. Ez a valóságban igen ritkán történik így. Sokkal gyakoribb, hogy az objektum jellemzőinek az objektum létrehozásakor adunk értéket.

Az értékadásra egy speciális függvény, az `__init()__` szolgál. Az `__init__()` függvény a példányosításkor automatikusan lefut, és végigcsinálja mindazt, amit beleírtunk. Úgy mondjuk, hogy ilyenkor történik meg az objektum inicializálása (magyarra előkészítésnek fordíthatnánk, de nem szokás fordítani). Néhány más programozási nyelvben az osztályok, objektumok hasonló szerepű függvényét konstruktornak nevezik, és ez az elnevezés olykor átszivárog a Python világába is.

Az osztály definíciója a következőképp módosul:

A függvény az osztály belsejében van.

```
1. class Diák:
2.     def __init__(self, vn, kn, évf):
3.         self.vezetéknév = vn
4.         self.keresztnév = kn
5.         self.évfolyam = évf
```

Az __init__() függvény első paramétere hagyományosan a self, és azt mondjuk el vele, hogy magán a függvényt tartalmazó objektumon dolgozzon a függvény.

Ezek az utasítások a függvény belsejében vannak.

A self miatt ezek a változók vagy jellemzők az objektum saját jellemzői lesznek. (Létrehozhatnánk olyan jellemzőket is, amelyek nem az egyes objektumokhoz, hanem az osztályhoz tartoznak, azaz minden, az osztályba tartozó objektum közös jellemzői.)

Az egyes objektumok példányosítása megváltozik:

Ezeket a paramétereket az __init__() függvény kapja meg. A „self”-et nem kell ideírnunk.

```
egyik = Diák('Lopakodó', 'Kasztanyetta', 11)
másik = Diák('Surranó', 'Szalicil', 11)
```

A többi utasítást pedig pontosan úgy használhatjuk, mint eddig. Próbáljuk is ki őket! Természetesen létre tudunk hozni új objektumot akár billentyűzetről, akár fájlból bekért adatok alapján:

```
v = input('Mi a harmadik diák vezetékneve? ')
k = input('Mi a harmadik diák keresztneve? ')
é = int(input('Mi a harmadik diák évfolyama? '))
harmadik = Diák(v, k, é)
```

Az általunk létrehozott objektumok is elhelyezhetők szótárban vagy – mint a példa mutatja – listában:

```
diákok = [egyik, másik, harmadik]
print(diákok[-1].keresztnév)
```

Feladatok

1. Írjunk programot `allataink.py` néven! Írjuk meg az `Állat` osztályt és az `__init__()` függvényét! A függvény a paraméterei alapján állítsa be jellemzőként az egyes példányok nevét, faját és korát!
2. Állítsunk elő `Állat` osztályú objektumokat az `allatok.txt` fájl alapján! Az objektumokat tároljuk az állatok nevű listában!
3. Járjuk be a listát, és írjuk ki a malacok nevét!

```
1. class Állat:
2.     def __init__(self, név, faj, kor):
3.         self.név = név
4.         self.faj = faj
5.         self.kor = kor
6.
7. állatok = []
8. with open('allatok.txt') as forrásfájl:
9.     for sor in forrásfájl:
10.        sor = sor.strip().split()
11.        állat = Állat(sor[0], sor[1], int(sor[2]))
12.        állatok.append(állat)
13.
14. for állat in állatok:
15.     if állat.faj == 'malac':
16.         print(állat.név)
```

Az állat és az Állat két különböző dolog a kis- és a nagybetű miatt. Az állat egy Állat osztályú objektum.

Ha elég szemfülesek vagyunk, észrevehettük, hogy a programot egyszerűbben, osztályok használata nélkül is megírhattuk volna (sőt, már írtunk is hasonlót pár leckével korábban):

```
állatok = []
with open('allatok.txt') as forrásfájl:
    for sor in forrásfájl:
        sor = sor.strip().split()
        állat = { 'név': sor[0], 'faj': sor[1], 'kor': int(sor[2]) }
        állatok.append(állat)

for állat in állatok:
    if állat['faj'] == 'malac':
        print(állat['név'])
```

Szótárakat hozunk létre, nem objektumokat.

A kiírás picit változik.

A Python szótár adattípusa remekül használható objektumok tárolására. Egy-egy objektumnak egy-egy szótár felel meg, a jellemzők nevei a szótár kulcsai, a jellemzők értékei a szótár kulcsaihoz tartozó értékek.

Akkor hát mi értelme osztályokkal meg objektumokkal vesződni? Két lecke múlva kiderül.

Az objektumokat tároló csoportnapló

Ebben a leckében az a feladatunk, hogy írjuk át a csoportnaplós programunkat, mégpedig úgy, hogy ne szótárakban, hanem saját magunk által írt osztályba tartozó objektumokban tárolja az egyes diákok adatait.

Tanulmányozzuk egy kicsit a programunk első kész változatát. Látható, hogy a fájl beolvasása, illetve egy-egy sor megfelelő darabolása elég jelentős része a programunknak. Megtehetjük-e, hogy az egyes sorokat már az osztályunk `__init__()` függvényével értelmeztetjük? Hát persze!

```
1. def átlag(rendes_jegyek, témazáró_jegyek):
2.     összeg = sum(rendes_jegyek) + sum(témazáró_jegyek) * 2
3.     osztó = len(rendes_jegyek) + len(témazáró_jegyek) * 2
4.     return round(összeg/osztó, 2)
5.
6. class Diák():
7.     def __init__(self, sor):
8.         self.név = sor[0]
9.         self.osztály = sor[1]
10.        jegyek = sor[2].split()
11.        self.rendes_jegyek = []
12.        self.témazáró_jegyek = []
13.        for jegy in jegyek:
14.            if len(jegy) == 1:
15.                self.rendes_jegyek.append(int(jegy))
16.            else:
17.                self.témazáró_jegyek.append(int(jegy[0]))
18.
19. diákok = []
20. with open('naplo.txt') as forrásfájl:
21.     for sor in forrásfájl:
22.         sor = sor.strip().split(', ')
23.         diák = Diák(sor)
24.         diákok.append(diák)
```

A függvény nem változik.

Ez a ,,sor'' már egy háromtagú lista.

A ,,jegyek'' és a ,,jegy'' változó ideiglenes, csak a függvényen belül létezik. Ezért nem kell eléjük self.

Amelyik változónak meg kell maradnia az __init__() lefutását követően is, annak self kerül a neve elé. Objektumváltozók, azaz jellemzők lesznek belőlük.

A fájlból beolvasott sort vessző + szóköz mentén daraboljuk.

Objektumot példányosítunk.

A kész objektumot a lista végére illesztjük.

A darabok listáját megkapja az osztály __init__() függvénye, amely a listában lévő adatok alapján feltölti az objektum jellemzőt.

```
25.
26. print('11. évfolyamra jár:')
27. for diák in diákok:
28.     if diák.osztály[:2] == '11':
29.         print(diák.név)
30.
31. print('Nem írt meg minden témazárót:')
32. for diák in diákok:
33.     if len(diák.témazáró_jegyek) < 3:
34.         print(diák.név, ': ', ', '.join(map(str, diák.témazáró_jegyek)), sep='')
35.
36. legkevesebb_szám = 10
37. legkevesebb_név = None
38. for diák in diákok:
39.     if diák.osztály == '11x' and len(diák.témazáró_jegyek) < legkevesebb_szám:
40.         legkevesebb_szám = len(diák.témazáró_jegyek)
41.         legkevesebb_név = diák.név
42. print(legkevesebb_név, 'írta a legkeveseeb témazárót a 11. x osztályban.')
43.
44. legtöbb_rendes_jegy = 0
45. for diák in diákok:
46.     if len(diák.rendes_jegyek) > legtöbb_rendes_jegy:
47.         legtöbb_rendes_jegy = len(diák.rendes_jegyek)
48. print('Felelésre számíthat:')
49. for diák in diákok:
50.         if len(diák.rendes_jegyek) < legtöbb_rendes_jegy * 0.8:
51.             print(diák.név)
52.
53. for diák in diákok:
54.     á = átlag(diák.rendes_jegyek, diák.témazáró_jegyek)
55.     if á <= 1.7:
56.         jegy = 1
57.     elif 1.7 < á < 4.5:
58.         jegy = int(round(á, 0))
59.     else:
60.         jegy = 5
61.     print(diák.név, 'átlaga:', á, 'jegye:', jegy)
```

Nem a szótár kulcsára, hanem az objektum jellemzőjére hivatkozunk.

Nem a szótár kulcsára, hanem az objektum jellemzőjére hivatkozunk.

Objektumaink működni kezdenek – Függvények az objektumok belsejében

Két leckével ezelőtt láttuk, hogy amikor csak adatszerkezetként használjuk az objektumokat, akkor nem igazán lépünk előre ahhoz képest, amikor egy-egy „létező” tulajdonságait egy-egy szótárban tároljuk. Python nyelven programozva így nem is tudtuk jelentőségének megfelelően értékelni ezt a lehetőséget. Sok más nyelvben nincs szótár vagy ahhoz hasonló képességű beépített adattípus, és már az adatszerkezetet megvalósító objektumot sem éreztük volna elhanyagolható előrelépésnek.

Az objektumok ereje ott mutatkozik meg igazán, amikor az osztályon, a terven belül függvényeket is írunk. Itt az ideje, hogy definíciót adjunk: **az objektumosztály egy adatszerkezet és az adatszerkezeten végezhető műveletek együttese.**

A „műveletek”-et elég tágan kell értenünk, mint az hamarosan kiderül.

31. példa: A macskák fejlődésének nyomon követése

Egy `macsek.py` nevű programfájlban hozzunk létre egy `Macska` nevű osztályt! A belőle példányosított objektumok az objektum létrehozásakor paraméterként kapják meg a macska nevét, születési évét és grammban kifejezett tömegét. Az `__init__()` függvény ezenfelül hozza létre a `tapasztalat` nevű jellemzőt, és adjon számára értékül egy 10 és 50 közötti véletlen számot! Ezzel a jellemzővel követjük majd nyomon a macskánk fejlődését. Minden sikeres vadászat növeli a tapasztalatot és ezzel a következő vadászat sikerének esélyét.

```
1. import random
2.
3. class Macska:
4.     def __init__(self, név, év, tömeg):
5.         self.név = név
6.         self.születési_év = év
7.         self.tömeg = tömeg
8.         self.tapasztalat = random.randint(10, 50)
```

Az osztályból objektumokat például a

```
cs = Macska('Csilus', 2018, 3652)
z = Macska('Zokni', 2017, 4003)
```

parancsok példányosítanak.

Az osztály műveleteit függvényekkel (eljárásokkal) valósítjuk meg. Már láttunk is egy ilyen függvényt, nevezetesen az `__init__()` nevűt. Az osztályok, objektumok belsejében létező függvényeket ebben a könyvben szinte mindig **tagfüggvénynek** hívjuk – már találkozhattunk a szóval. Ez a név kifejezi, hogy ezek is függvények. Talán a legelterjedtebb elnevezésük a metódus, de szokás őket függvénytagnak is hívni.

Az első tagfüggvényünk arra való, hogy a macskáink „nyávogni” tudjanak. A tagfüggvény egyetlen paramétere az objektumra visszautaló `self`. Ahogy az `__init__()` esetén már láttuk, híváskor ezt a paramétert nem kell átadnunk.

```
 9.
10.     def nyávog(self):
11.         print('Nyaú!')
```

A tagfüggvényt az alábbi módon hívjuk:

```
cs.nyávog()
z.nyávog()
```

A második tagfüggvényünk azt szemlélteti, hogy a tagfüggvényeknek is lehet visszatérési értékük. A példában szereplő tagfüggvény egy jellemzőt is felhasznál a visszatérési érték kiszámításához.

```
12.
13.     def kor(self, mostani_év):
14.         return mostani_év - self.születési_év
```

A tagfüggvény használata 2022-ben:

```
print(cs.kor(2022))
```

A harmadik tagfüggvény arra mutat példát, hogy miként lehetséges egy jellemző értékének megváltoztatása tagfüggvény hívásával:

```
15.
16.     def eszik(self):
17.         self.tömeg += 25
```

A tagfüggvény hívásakor nem kell megadnunk paramétert:

```
print(cs.tömeg)
cs.eszik()
print(cs.tömeg)
```

Természetesen egy tagfüggvény többször is hívható:

```
print(z.tömeg)
for _ in range(10):
    z.eszik()
print(z.tömeg)
```

Az osztály utolsó előtti tagfüggvénye pedig azt példázza, hogy egy tagfüggvény egészen bonyolult műveleteket is megvalósíthat.

Felbukkan a véletlen tapasztalatú egér…

Programunk eposzi seregszemlét nyújt a harc kezdete előtt.

```
18.
19.     def egerészik(self):
20.         egér_tapasztalata = random.randint(1, 100)
21.         print(self.név, 'egy', egér_tapasztalata, 'tapasztalatú egérrel próbálkozik.')
22.         if self.tapasztalat > egér_tapasztalata:
23.             print('Megfogta!')
24.             if self.tapasztalat < 100:
```

Aki tapasztaltabb, az nyer.

```
25.                 self.tapasztalat += 1
26.             self.eszik()
27.         else:
28.             print('Elszaladt :(')
```

A sikeres vadászat növeli a tapasztalatot, legfeljebb 100-ig…

…és tömegnövekedést is okoz – egy másik tagfüggvény hívásával.

Kövessük végig valamelyik macskánk egerészéseinek egy hosszabb sorozatát:

```
for _ in range(50):
    print(z.név, 'tömege:', z.tömeg, ', tapasztalata:', z.tapasztalat)
    z.egerészik()
```

Mielőtt az osztály utolsó tagfüggvényének megvalósításába fognánk, itt az ideje szembesülnünk azzal, hogy Pythonban programozva jószerével az első pillanattól kezdve objektumokat használunk. Anélkül, hogy túlzottan mélyen elmerülnénk a témában, lássunk néhány, az előző kijelentést illusztráló programsort, melyeket akár mostani programunk végére is beírhatunk. Megjegyezzük, hogy a „típus” és az „osztály” fogalma többé-kevésbé felcserélhető, például lista típusú adat helyett mondhatunk lista osztályú objektumot is.

A type() függvény megmondja, hogy egy adott objektum melyik objektumosztályba tartozik.

<class '__main__.Macska'> típusú/osztályú objektum

```
print('A z objektum típusa:', type(z))
```

A type() függvény megmondja, hogy egy adott objektum melyik objektumosztályba tartozik.

<class 'int'> típusú/osztályú objektum

```
szám = 5
print('Az 5 típusa:', type(szám))
szám = int(5)
print('Az int(5) típusa:', type(szám))
```

Ilyenkor az int() osztály objektuma az 5 értékkel inicializálódik.

A két sor ugyanolyan listát, ha úgy tetszik, list osztályú objektumot hoz létre.

```
lista1 = [1, 2]
lista2 = list((1, 2))
print('A listák típusai:', type(lista1), type(lista2))

szótár1 = {'kulcs1': 'érték1', 'kulcs2': 'érték2'}
szótár2 = dict(kulcs1 = 'érték1', kulcs2 = 'érték2')
print('A szótárak típusai:', type(szótár1), type(szótár2))
```

A két sor ugyanolyan szótárat, ha úgy tetszik, dict osztályú objektumot hoz létre.

A lecke elején említett definíció második részében („és az adatszerkezeten végezhető műveletek”) arról van szó, hogy milyen műveleteket megvalósító tagfüggvényei vannak egy osztálynak. Ismerjük például a `list` osztály `append()` tagfüggvényét, a `dict` osztály `items()` tagfüggvényét vagy az `str` osztály `strip()`, `split()` és `lower()` tagfüggvényeit.

Utolsó tagfüggvényként egy speciális Python-függvényt valósítunk meg. Pythonban programozva természetesnek vesszük, hogy az egyes objektumoknak van szöveges reprezentációjuk: ki tudjuk adni a `print(szótár1)` parancsot, és értelmezhető kimenetet kapunk, ami különösen a kódolás hibakeresési fázisában hasznos. Mindez saját magunk készítette osztályok objektumaival alapesetben nem működik. Próbáljuk ki!

```
print(z)
```

Nem azt látjuk, aminek igazán örülnénk. Megjelenik ugyan egy, az objektum memóriában való elhelyezkedésével kapcsolatos információ, de jobb lenne, ha látnánk is, hogy mi van az objektumban. Szerencsére mindössze annyi a dolgunk, hogy megvalósítunk egy újabb tagfüggvényt a `Macska` osztályban:

```
29.
30.     def __str__():
31.         return self.név + ' születési éve: ' + str(self.születési_év) + \
32.             ', tömege: ' + str(self.tömeg) + ' g.'
```

Ha egy objektumnak van `__str__()` tagfüggvénye, akkor annak a visszatérési értékét írja ki a `print()` függvény. A Python beépített adattípusai, objektumosztályai mind a fentihez hasonló módon biztosítanak lehetőséget arra, hogy könnyen értelmezhető formában megismerjük az objektum tartalmát. Ha szeretnénk olyan objektumokat készíteni, amelyek minden szituációban a Python beépített objektumaihoz hasonlóan viselkednek, a *magic methods* vagy a *dunder methods* kifejezésre érdemes keresnünk az interneten.

Feladatok

1. Oldjuk meg, hogy macskáink véletlenszerűen nyávogjanak „Nyaú!”-t vagy „Mijaú!”-t! A `Macska.eszik()` tagfüggvény helyett alkossuk meg a `Macska.tápot_eszik()` és a `Macska.egeret_eszik()` tagfüggvényt! Az előző vegye át a régi függvény szerepét, az új pedig legyen paraméterezhető a megevett egér grammban kifejezett tömegével! A megevett egér tömege véletlenszerűen 5–25 százalékban fordítódjék a macska tömegének növelésére! A `Macska.egerészik()` tagfüggvényt módosítsuk úgy, hogy az „egereknek” ne csak tapasztalatuk, hanem véletlenszerű (de a valóságban elképzelhető) tömegük is legyen! Az egér tömegét használjuk fel a `Macska.egeret_eszik()` tagfüggvény hívásakor!
2. Kihívást jelentő feladat: Két macska találkozásának gyakran elkeseredett nyávogás és az egyik fél pánikszerű menekülése a vége. De melyik macska fut el? A megfelelő mágikus tagfüggvények megvalósításával tegyük lehetővé, hogy két `Macska` osztályú objektum a szokásos relációs jelekkel összehasonlítható legyen! Az a `Macska` legyen „nagyobb”, amelyiknek a tapasztalata nagyobb!

Működésben a csoportnapló objektumai

Négy leckével ezelőtt *(Csoportnapló – Tapogatózás az objektumorientált programozás irányába)* írtuk meg a csoportnaplós programunkat először. Akkor még egyáltalán nem használtunk objektumokat, a programunk eljárásközpontú volt. Nem volt rosszabb, mint a programunk második, illetve most elkészülő harmadik változata, de azért az akkori töprengéseink során láttuk, hogy komoly lépéseket kell tennünk, ha azt szeretnénk, hogy a programunk könnyebben áttekinthető, karbantartható és továbbfejleszthető legyen.

A program második változata már használ objektumokat, de ezek az objektumok még csak adatszerkezetek: az adatokkal dolgozó függvények még az objektumokon kívül vannak, azaz még nem tagfüggvények.

Most elkészítjük a program harmadik változatát. Átgondoljuk, hogy melyek azok a függvények, amelyek egy-egy diákkal vagy a diák jegyeivel dolgoznak: azok, amelyek megjelenítik vagy visszatérési értékükben megadják a szóban forgó diák egy-egy jellemzőjét, vagy amelyek műveleteket végeznek a diák valamelyik jellemzőjével.

Az ilyen függvényeket tagfüggvényekké alakítjuk, és megvalósítjuk azokat a lehetőségeket, amelyeket töprengéseink során hasznosnak gondoltunk. A `Diák` osztály jelentősen kibővül:

```
1. class Diák():
2.     def __init__(self, sor):
3.         self.név = sor[0]
4.         self.osztály = sor[1]
5.         jegyek = sor[2].split()
6.         self.rendes_jegyek = []
7.         self.témazáró_jegyek = []
8.         for jegy in jegyek:
9.             if len(jegy) == 1:
10.                self.rendes_jegyek.append(int(jegy))
11.            else:
12.                self.témazáró_jegyek.append(int(jegy[0]))
13.
14.    def évfolyam(self):
15.        return int(self.osztály[:2])
16.
17.    def osztálybetű(self):
18.        return self.osztály[2:]
19.
20.    def osztály(self):
21.        return self.osztály
22.
```

Az __init__() tagfüggvény nem változik.

Ezt a két tagfüggvényt az első töprengésünkben álmodtuk meg.

Ez a tagfüggvény kényelmi szerepű: nem kell azon töprengeni, hogy a diák.osztály (egy jellemző) vagy a diák.osztály() (egy függvény) használatával tudjuk meg a diák osztályát.

```
23.    def megírt_minden_témazárót(self, idei_témazárók_száma):
24.        if self.jegyek_száma('témazáró') == idei_témazárók_száma:
25.            return True
26.        else:
27.            return False
28.
```

Megadhatjuk, hogy hány témazárót írt idén a csoport, így programunk használhatóságát növeljük.

Ha szeretnénk úgy megírni a tagfüggvényt, hogy csak egy `return` utasítás szerepeljen benne, akkor az utolsó négy sorát helyettesítsük ezzel az eggyel:

```
return self.jegyek_száma('témazáró') == idei_témazárók_száma
```

A második töprengésben olyan tagfüggvényt szerettünk volna, amelyiknek a hívási kódját olvasva azonnal látható, hogy mit csinál a program. Az alábbi egy kicsit ügyesebb az akkor elképzeltnél: többféle jegytípust kezel. Ráadásul bővíthető, ha egyszer majd megkülönböztetjük az órai munkára kapott jegyeket. A jegy típusát a `milyen` paraméterben adhatjuk át a tagfüggvénynek.

```
29.     def jegyek_száma(self, milyen):
30.         if milyen == 'témazáró':
31.             return len(self.témazáró_jegyek)
32.         elif milyen == 'rendes':
33.             return len(self.rendes_jegyek)
34.         elif milyen == 'összes':
35.             return len(self.témazáró_jegyek) + len(self.rendes_jegyek)
36.
```

Az utolsó töprengésünknek megfelelően az `átlag()` függvényt úgy alakítjuk át, hogy annak a programozónak, aki használja, ne kelljen tudnia, hogy most épp milyen szabályok szerint számolunk átlagot. Ha az átlagszámítás módja változik, akkor csak ezen az egy helyen kell belenyúlnunk a programba.

```
37.     def átlag(self):
38.         összeg = sum(self.rendes_jegyek) + sum(self.témazáró_jegyek) * 2
39.         osztó = self.jegyek_száma('összes') + self.jegyek_száma('témazáró')
40.         return round(összeg/osztó, 2)
41.
```

Az év végi jegy végső soron a diák tulajdonsága, azaz az objektumosztályban van a helye a jegyet visszaadó függvénynek.

```
42.     def év_végi_jegy(self):
43.         á = self.átlag()
44.         if á <= 1.7:
45.             return 1
46.         elif 1.7 < á < 4.5:
47.             return int(round(á, 0))
48.         else:
49.             return 5
```

A tagfüggvény egy másik tagfüggvényt hív a jegy megállapításakor.

Elkészültünk a `csoportnaplo.py` végső változatával. Persze az objektumorientált programozás témakörét messze nem merítettük ki, szédítő mélységeibe épp csak lepillantottunk. Ahhoz azonban eleget tanultunk, hogy – egy szükséges kitérőt követően – grafikus felhasználói felületű programokat írhassunk.

Egy szükséges vargabetű: kulcsszó-paraméteres függvények és modulok

Ebben a leckében a Python nyelvű programok két olyan, szorosan nem összefüggő részterületével foglalkozunk, amelyek szükségesek ahhoz, hogy értsük a grafikus felhasználói felület programozását. Ugyanakkor mindkét új tudományunk remekül használható a programozás minden más területén is.

Kulcsszó-paraméteres függvények

Futtassuk le az alábbi kódot!

```
1. def köszön(hogyan, kinek):
2.     return hogyan + ' ' + kinek + '!'
3.
4. print(köszön('Ave,', 'Caesar'))
5. print(köszön('Szevasz,', 'Tavasz'))
```

A fenti függvénnyel az a probléma, hogy ha esetleg felcseréljük a paramétereit, akkor nem úgy köszön a programunk, ahogyan szeretnénk. A veszély fokozódik az olyan függvényeknél, amelyek sok paramétert várnak.

A függvények fenti paraméterezése az úgynevezett **helyzeti** vagy pozicionális **paraméterezés**. Azért nevezzük így a módszert, mert a függvény az átadott paraméterek egymáshoz viszonyított helyzetéből, sorrendjéből tudja, hogy melyikkel mi a teendője.

Írjuk át az első sort:

```
1. def köszön(hogyan='Ave,', kinek='Caesar'):
```

Így a függvényünk **kulcsszóparamétereket** (magyarul gyakran fordítjuk nevesített paraméternek is az angol keyword parameter kifejezést) kapott, azaz a paramétereknek kulcsszavuk, nevük van. Az ilyen függvényeknél egyértelmű a paraméterek jelentése, a sorrendjük pedig többé nincs erősen kötve. Ha a kulcsszó-paraméteres függvénynek híváskor nem adjuk meg a paramétereit, akkor a függvénydefinícióban lévő, alapértelmezett paramétert használja a függvény.

Az alábbiakban néhány kódsort, kódsorpárt közlünk, azokat követőleg pedig igaz megállapításokat a kulcsszóparaméterek használatáról. A kódsorok nem minden esetben futnak le helyesen. A feladatunk azt megállapítani, hogy melyik kódrészlet futtatásából melyik megállapítás következik.

```
1)  print(köszön('Szevasz,', 'Tavasz'))
2)  print(köszön())
3)  print(köszön(hogyan='Szevasz,'))
    print(köszön(kinek='Tavasz'))
4)  print(köszön(hogyan='Szevasz,', kinek='Tavasz'))
5)  print(köszön(kinek='Tavasz', 'Ave,'))
6)  print(köszön('Ave,', kinek='Tavasz'))
7)  print(köszön(hogyan='Szevasz,', 'Tavasz'))
```

A) Továbbra is megadhatjuk a paramétereket helyzeti paraméterként.
B) Nem kell minden paramétert megadni. Amelyiket nem adjuk meg, annak az alapértelmezett értékét használja a függvény.
C) Paraméterek nélkül a függvény a definíciókor megadott alapértelmezett értékeket használja.
D) A kulcsszóparaméterek megadási sorrendje felcserélhető.
E) Kulcsszóparamétert nem követhet helyzeti paraméter, még akkor sem, ha jó a paraméterek sorrendje.
F) Kulcsszóparamétert nem követhet helyzeti paraméter.
G) Helyzeti paramétert követhet kulcsszóparaméter, például `print('Szia', 'mia!', sep='-')`

A kulcsszóparaméterek értéke megadható változóval is, így néha az alábbi, első találkozáskor még meglepő függvényhívást látjuk:

```
hogyan = 'Pá,'
kinek = 'kis aranyom'
print(köszön(hogyan=hogyan, kinek=kinek))
```

Ez a paraméter kulcsszava, neve.

Ez a változó neve.

Modulok

Ha eddig szigorúan tartottuk magunkat a tankönyvben írtakhoz, akkor egyetlen modult importáltunk, éspedig a random nevűt. A modul függvényei közül a `randint()` nevűvel véletlen egész számokat állíthatunk elő, a `choice()` pedig arra jó, hogy például a számára átadott lista véletlenszerűen kiválasztott elemét visszaadja.

A modulok arra valók, hogy bizonyos, gyakran használt, előre megírt programrészleteket tároljunk bennük.

Vannak olyanok, amelyek az alap Python-telepítéssel együtt kerülnek a gépünkre, és sok-sok olyan is van, amelyeket külön kell telepítenünk. Az informatika számos – de nem erős túlzás a minden szó használata sem – területével kapcsolatosan léteznek modulok. Van hálózatot kezelő, hangfájlokat tölteni-menteni képes vagy 3D-modellek készítésére alkalmas modul; van olyan, amellyel a közösségi médiára posztolhatunk automatikusan; olyan, amellyel robotot irányíthatunk, játékprogramot fejleszthetünk vagy mesterséges intelligenciát alkothatunk, és még ki tudja, hányféle. Ilyen modulokban kapnak helyet azok az előre elkészített programrészek, amelyekkel a grafikus felhasználói felületű programok írása igen jelentős mértékben egyszerűsödik, és amilyet a következő leckétől kezdve mi is használunk.

A modulok a legegyszerűbb esetben nem mások, mint Python-fájlok, amelyek vagy a fő programfájlunkkal közös mappában, vagy olyan egyéb mappában vannak, ahol a gépünkre telepített Python-parancsértelmező keresi őket. Ezekben a fájlokban nem egész programok találhatók, hanem például változók, függvények vagy osztályok definíciói.

32. példa: Hazánk békafajainak listája

Készítsünk mi is egy modult! Hozzunk létre egy fájlt `beka.py` néven, benne egy, a magyarországi békafajokat felsoroló listával:

```
1. békák = ['vöröshasú unka', 'sárgahasú unka',
2.          'barna ásóbéka', 'barna varangy',
3.          'zöld varangy', 'zöld levelibéka',
4.          'gyepi béka', 'mocsári béka',
5.          'erdei béka', 'tavi béka',
6.          'kis tavibéka', 'kecskebéka']
7.
```

Amikor elkészültünk, nyissunk egy másik fájlt a szerkesztőnkben, és mentsük azt `foprogram.py` néven (a `beka.py`-t még ne zárjuk be)! Az új fájlunknak egyelőre csak három sora lesz:

```
1. import beka
2.
3. print(', '.join(beka.békák))
```

Importáljuk a saját modulunkat.

Az új fájlt mentsük a `beka.py` fájlnéven egy mappába, majd futtassuk!
Ha minden jól ment, azt látjuk, hogy kiíródtak a békafajok nevei. Láttuk, hogy a modulban lévő listára hasonlóan hivatkozunk, mint a `randint()` függvényre szoktunk: a lista neve előtt megadjuk a modul nevét is. Szakkifejezést használva úgy fogalmazunk, hogy a `békák` lista a `beka` *névtérben* van, és az elérésekor megadjuk a névteret is.

Természetesen írhatunk függvényt is a modulunkba. Írjunk egy olyan függvényt, amely a lista valamelyik véletlen elemét adja vissza! Szükség lesz hozzá a random modulra, amelyet kivételesen nem a `beka.py` fájl elején importálunk, mert itt a könyv lapjain utólag nehézkes átszámozni a sorokat. Bővítsük a `beka.py` fájlt:

```
8. import random
9. def véletlen_béka():
10.     return random.choice(békák)
11.
```

Egy modul használhat más modulokat.

Ha mentettük a fájlt, a főprogramban használatba vehetjük új függvényünket, például így:

```
print(beka.véletlen_béka())
```

Hozzunk létre egy osztályt is a modulunkban, a `beka.py` fájlban:

```
12. class Béka():
13.     def __init__(self):
14.         self.faj = véletlen_béka()
15.
16.     def hangot_ad(self):
17.         hang = random.choice(['Brekk!', 'Kurutty!'])
18.         return hang
```

A modulban tudjuk használni a modul egyéb függvényeit.

Az osztályt használatba vehetjük a főprogramból:

```
kedvenc = beka.Béka()
print(kedvenc.faj, 'megszólal:', kedvenc.hangot_ad()
```

Végezetül röviden érintünk még két érdekességet. Tesszük ezt azért, mert az interneten kóborolva mindkettővel gyakran találkozhat a Pythonban programozó programozó, és fontosnak érezzük, hogy értse, amit lát.

Lehetőség van nem teljes modulok, hanem modulrészek importálására is. A

```
from beka import Béka
```

parancs csak a Béka osztályt importálja a modulból, amelyet ezt követően nem `beka.Béka`, hanem egyszerűen `Béka` néven használhatunk. A módszer előnye a rövidebb kód, hátránya, hogy ha a főprogramban is használjuk a Béka nevet, akkor bajba kerülünk, mert az egyik név elfedi, és így használhatatlanná teszi a másikat.

Akár a modulok, akár a modulrészek hivatkozhatók az eredetitől eltérő néven importálást követően. Egész modul esetén

```
import modul as nálunk_használt_név
```

modulrész esetén pedig

```
from modul import rész as nálunk_használt_név
```

a megfelelő utasítás. A módszert olyankor használhatjuk, ha az eredeti név hátráltat bennünket: például használjuk már valami másra, vagy egyszerűen csak túl hosszú.

Grafikus felhasználói felületű alkalmazást fejlesztünk – A Sajáttömb

A programozás témakör hátralévő leckéiben két alkalmazást készítünk el. Mindkettő grafikus felhasználói felülettel (angol szakkifejezéssel: graphical user interface, GUI) rendelkezik.

Hogyan készül egy GUI?

A grafikus alkalmazások fejlesztése az esetek túlnyomó többségében valamilyen mások által elkészített keretrendszer, programrészleteket tartalmazó szoftverkönyvtár használatára támaszkodik. Ezek a keretrendszerek a grafikus felület számos összetevőjét, részét készen bocsátják rendelkezésünkre, azaz nem nekünk kell megírni például azt, hogy

- legyenek ablakaink, amelyek az operációs rendszer grafikus felületével együttműködnek;
- az ablakokon meglegyen a szokásos három gomb (kicsinyítés, teljes képernyő, bezárás);
- az ablakok az egérrel mozgathatók legyenek;
- az ablakok szélét az egérrel megfogva át lehessen őket méretezni;
- és még számos egyéb részt, amelyek közül néhánnyal megismerkedünk alkalmazásaink elkészítése során.

Az egyes keretrendszerek használata, programozása többé-kevésbé hasonló, egyes területeken pedig nagyon eltér egymástól. De ha már bármelyikben programot írunk, akkor elég jó elképzeléseink lesznek a grafikus alkalmazások fejlesztésének folyamatáról, és ezt bármely más környezetben is kamatoztathatjuk.

Vannak olyan keretrendszerek, amelyeket csak egyetlen nyelven fejlesztve tudunk használni, és van, amelyet többből is. Vannak olyanok, amelyek egyetlen operációs rendszeren, grafikus felületen működnek, és vannak, amelyek többfélén.

Pythonban programozva nagyon sok grafikus keretrendszer áll rendelkezésünkre. Az általunk használt **Tk környezet** az alap Python-telepítés részét képezi, a rá támaszkodó programok változtatás nélkül futnak a három nagy asztali operációs rendszeren. Ez a környezet is nagyon **sok konstanst, változót, objektumot, tagfüggvényt tartalmaz,** amelyek nevét, paraméterezését, működését a szakirodalmi leírásokból lehet megismerni. A tárgyalásunkban most ezeket megadjuk.

A Python a Tk környezettel egy erre szolgáló felületen (angolul: interface) kommunikál. A felületet `tkinter` néven modulként importáljuk. A legegyszerűbb grafikus program az alábbi három (öt) sorból áll:

A főciklus

A főciklus minden grafikus felületű program lényegi része. Elindítása előtt kialakítottuk a programunk fő ablakát, de ekkor még nem jelenik meg az ablak. A főciklus elindításakor az ablak megjelenik, és a programunk hozzákezd ahhoz a tevékenységhez, amit a futása alatti idő túlnyomó részében tenni fog: várakozik.

A főciklusban lévő program egy, a gazdája mellett ácsorgó inashoz hasonlítható, aki a gazdája minden szavát figyeli, és ha parancsot kap, rohan végrehajtani. A programunk a „gazdának", azaz a számítógép felhasználójának „parancsairól", billentyűlenyomásairól, egérkattintásairól, azaz a bemeneti perifériákról érkező jelzésekről úgynevezett **események** formájában kap értesítést. A perifériák figyelése az operációs rendszer dolga. Az operációs rendszer veszi észre, ha a felhasználó lenyomott egy gombot, mozgatta az egerét, vagy kattintott vele, illetve az érintőképernyőt használta, és ha a mi programunk van „elöl" (szaknyelven: fókuszban), akkor továbbítja neki az eseményt. A programunk pedig úgy reagál az eseményre, ahogy megírjuk a programkódban.

A Sajáttömb

Az első alkalmazásunk egy, a mindenki által ismert Jegyzettömbhöz hasonló program, a *Sajáttömb* lesz. Írjuk be a fenti három kódsort egy `sajattomb` nevű programba, és futtassuk a programot – akár a fájlkezelőből – dupla kattintással! (Windowson a grafikus felülettel működő Python programoknak érdemes `.pyw` kiterjesztést adni.)

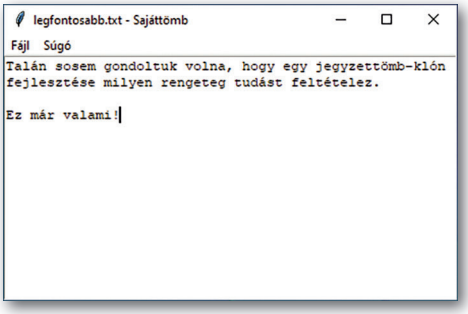
Látjuk, hogy a programunknak van ablaka, rendelkezik címsorral, az azon lévő nyomógombokkal, és átméretezhető akár a gombok használatával, akár az ablak szélét egérrel megfogva. Mindezekért egyáltalán nem dolgoztunk meg a kódunkban – valaki más, a modul írója dolgozott helyettünk. Szempontunkból a programunk „magától" képes ezekre a mutatványokra. Talán ezen a ponton kezdjük el igazán megérteni, hogy milyen előnyökkel jár a grafikus keretrendszerek használata, illetve azt, hogy egy-egy modul betöltése miként növeli ugrásszerűen a lehetőségeinket.

▶ A Sajáttömb Linuxon

Ha a programunkban van a főciklusba való belépést követő utasítás is, az egészen addig nem fut le, amíg a főablak be nem zárul.

Próbáljuk ki: írjunk ki egy szót a szokásos `print()` függvénnyel a főciklus sora után! A programot ezúttal ismét érdemes parancssorból futtatni. A fejlesztés alatt egyébiránt minden grafikus programot érdemes a parancssorból futtatni, mert itt látjuk majd a program hibaüzeneteit is.

A grafikus alkalmazások fejlesztése során nem ritka, hogy az alkalmazás egy önálló osztályba kerül, és a programunk lényegében az osztályból példányosított egyetlen objektumként fut. Ennek többek között az objektumjellemzők



▶ A Sajáttömb Windowson

egyszerűbb elérése az oka: az objektum tagfüggvényei az objektumjellemzőket akkor is egyszerűen tudják módosítani, ha azok a függvényen kívül is léteznek. Mi is kihasználjuk ezt az előnyt. Alakítsuk ilyenné a programunkat:

```
1. import tkinter
2.
3. class App:
4.     def __init__(self, master):
5.         self.master = master
6.
7. főablak = tkinter.Tk()
8. app = App(főablak)
9.
10. főablak.mainloop()
```

Lényegében az egész program itt kap helyet.

Az objektum megyjegyzi, hogy ki a „főnök”, azaz hogy melyik ablak belsejében fut. Ez később még jól jöhet. A paraméter neve hagyományosan master, de átnevezhetjük.

Itt van az egyetlen példányosítása az osztálynak. A paraméterben megmondjuk, hogy a főablak belsejében fusson a program.

A programunkat ezen a ponton lefuttatva nem sok változást látunk, de a főablakban már fut a programunk lényegi része – csak még nem csinál semmit.

A szövegmező

A programunk írását azzal folytatjuk, hogy elkészítjük és megjelenítjük a jegyzettömbünk lényegi részét, azt a szövegmezőt, ahová a szöveget lehet írni. Tapasztalatból tudjuk, hogy az ilyen szövegmező sok mindenre képes. Lássunk néhányat a képességei közül:

- Kurzorral jelzi, hogy hova kerül a következő karakter. A kurzor helyzetét a tevékenységeinknek megfelelően frissíti.
- Ha a billentyűzetről karaktert kap (egy *esemény* történik), akkor a karaktert megjeleníti.
- Enter lenyomására új sort kezd, a kurzort a sor elejére teszi.
- A sor végét elérve új sort kezd.
- A szokásos módon tudunk benne másolni és beilleszteni.

Mindezzel nagyon kevés dolgunk lesz, hiszen a keretrendszer megfelelő osztálya – a `Text` – már készen áll, nekünk csak példányosítani kell, és elhelyezni a példányt. Erre való az `App` osztályunkban elhelyezett alábbi tagfüggvény:

A Tk keretrendszer Text osztályából példányosítunk egyet magunknak.

```
    def szövegterületet_készít(self):
        self.szövegterület = tkinter.Text(self.master)
        self.szövegterület.pack(expand=True, fill='both')
```

A példányosításkor megmondjuk, hogy melyik ablakba kerüljön a szövegmező – ezért tettük el a főablakhoz vezető hivatkozást.

A pack() tagfüggvény hívása az egyik módszer egy grafikus objektum megjelenítésére a tkinterben. A paraméterei szerint nőjön akkorára, amekkorára lehet (expand), mindkét dimenzióban (both).

A fenti tagfüggvény hívására minden induláskor szükség van, úgyhogy helyezzük el az `__init__()` tagfüggvény végén ezt a sort:

```
self.szövegterületet_készít()
```

És indíthatjuk a programunkat. Igazán remekül működik!

A menüsor

A menüsorban két választási lehetőség lesz: a Fájl, illetve a Súgó. A menüket kialakító tagfüggvény:

```
def menüsort_készít(self):
    self.menüsor = tkinter.Menu(self.master)

    fájl_menü = tkinter.Menu(self.menüsor, tearoff=0)
    fájl_menü.add_command(label='Új')
    fájl_menü.add_command(label='Megnyitás')
    fájl_menü.add_command(label='Mentés')
    fájl_menü.add_separator()
    fájl_menü.add_command(label='Beállítások')
    fájl_menü.add_separator()
    fájl_menü.add_command(label='Kilépés', command=self.master.destroy)
    self.menüsor.add_cascade(label='Fájl', menu=fájl_menü)

    súgó_menü = tkinter.Menu(self.menüsor, tearoff=0)
    súgó_menü.add_command(label='Névjegy')
    self.menüsor.add_cascade(label='Súgó', menu=súgó_menü)

    self.master.config(menu=self.menüsor)
```

A Tk keretrendszer Menu osztályából példányosítunk egyet magunknak.

Létrehozzuk a Fájl menüt.

Összeállítjuk a Fájl menü tartalmát.

Elhelyezzük menüt a menüsorban.

Összeállítjuk a Súgó menüt.

Elhelyezzük a menüsorban.

A menüt nem lehet leválasztani a menüsorról.

Természetesen szükség van a tagfüggvény hívására az `__init__()` metódus végén:

```
self.menüsort_készít()
```

A programot elindítva látjuk, hogy a menü megjelenik, de nem működik, a *Kilépés* menüpontot kivéve. A kódot újra átnézve látjuk, hogy ennek az egy menüpontnak adtuk meg a `command`, azaz parancs paraméterét, ezért épp ez az egy működik. A menüpont parancsa a főablak `destroy()`, azaz 'megsemmisít' tagfüggvénye – ez fut le, ha a *Kilépés* menüpontot választjuk. A végrehajtandó függvényeket a `command` paraméterben a függvény végi zárójel nélkül kell megadnunk – azaz nem tudunk számukra paramétert átadni, és ebből még lesz problémánk.

A névjegy

Sok „igazi" programhoz hasonlóan a *Sajáttömb* is kap névjegyet, ahol például a program készítőit szokás megemlíteni. A névjegy új ablakának megtervezésével nem óhajtunk sok időt tölteni, egy rendelkezésre álló, úgynevezett üzenetablakot használunk.

Az első dolgunk a `tkinter` modulcsomag egy újabb moduljának importálása. A következő sort írjuk a meglévő `import` alá:

```
import tkinter.messagebox
```

Helyezzük el az osztályban az alábbi tagfüggvényt:

```
def ablak_névjegy(self):
    tkinter.messagebox.showinfo('Névjegy', 'Mi írtuk az egészet!')
```

Végül módosítsuk a *Súgó* menü megfelelő sorát, hogy futtassa az előbb elkészített tagfüggvényt:

```
súgó_menü.add_command(label='Névjegy', command=self.ablak_névjegy)
```

Vezérlőelemek, widgetek

Látjuk, hogy nagyon leegyszerűsítve egy grafikus felhasználói felület kétféle összetevőből áll: ablakokból és widgetekből (ejtsd: vidzsit). A widget szó a window + gadget (ablak + bigyó, esetleg ablak + kütyü) összevonás eredménye. Eddig két widgetet használtunk: a szövegmezőt `(Text)` és a menüsort `(Menu)`. További widgetek például a nyomógombok, a rádiógombok, a jelölőnégyzetek, a keretek, a gördítősávok és a legördülő menük; néhánnyal még megismerkedünk a könyvünkben.

A widgeteket szokás controlnak (magyarul vezérlőelemnek vagy csak vezérlőnek) is nevezni. Az, hogy melyik elnevezést használjuk, a szóban forgó keretrendszertől függ.

A grafikus programok eleje és vége

Mostanra látjuk, hogy egy grafikus programnak nincs eleje és vége abban az értelemben, ahogy az eddig írt programjainknak volt. Nem tudjuk megmutatni, hogy milyen sorrendben fognak lefutni az utasítások. Ha a *Sajáttömb* szövegterületére írunk, akkor az a widget aktív, ha a menüben bóklászunk, akkor pedig az. A program lényegében sok objektum együttese, melyek között mi teremtjük meg a kapcsolatot, és amelyek többnyire aszerint dolgoznak, hogy melyik milyen eseményről szerez tudomást a főciklus ismétlődései során.

A Sajáttömb fájlkezelése

Alkalmassá kell tennünk a programunkat a szerkesztett fájl mentésére, létező fájl megnyitására és új fájl kezdésére. Kezdjük ez utóbbival!

A program két esetben hoz létre új fájlt: az alkalmazás megnyitásakor, és amikor a megfelelő menüpontot választjuk. Lássuk azt a tagfüggvényt, amelyik mindkét esetben hívható:

Ha volt már ilyen változónk, ürítjük a tartalmát. Ha nem volt, mert most indult a program, akkor most létrehozzuk.

```
def fájl_üreset_állít(self):
    self.fájlnév = None
    self.szövegterület.delete('1.0', 'end')
```

Az első sor nulladik karakterétől a szövegterület végéig törlünk.

Helyezzük el a tagfüggvényt hívó sort az `__init__()` metódus végén, és adjuk meg az *Új* menü paramétereként is.

A második fájlkezelő tagfüggvényünk a fájl mentésére való. Mi is lesz a dolga?

- Ha még nincs fájlnév, akkor új fájlnevet kér.
- Ha van már fájlnév, azaz nem ez az első mentés, akkor azt használja.
- A fájlnévvel menti a szövegterület tartalmát – új fájlt ír, vagy felülírja az előző fájlt.

A programunknak nem lesz „mentés másként” lehetősége.

Az első dolgunk egy fájlválasztó párbeszédablak importálása az `import tkinter.filedialog` paranccsal.

A mentést végző tagfüggvény:

Ha még sosem volt mentve a fájl...

...akkor párbeszédablakot nyitunk, és bekérjük a fájlnevet.

Ha már volt mentve a fájl, akkor a meglévő nevet használjuk.

Ez az ellenőrzés azért kell, mert ha a párbeszédablakban Esc-et nyom a felhasználó, üres karakter lesz a fájlnév. Így Esc esetén tétlenek maradunk.

A célfájlba a szövegterület tartalma kerül.

```
def fájl_ment(self):
    if not self.fájlnév:
        fájlnév = tkinter.filedialog.asksaveasfilename(
                initialfile='szöveg.txt',
                defaultextension='.txt',
                filetypes=[('Szövegfájlok', '*.txt'),
                           ('Minden fájl', '*.*')])
    else:
        fájlnév = self.fájlnév
    if fájlnév != '':
        with open(fájlnév, 'w') as célfájl:
            célfájl.write(self.szövegterület.get('1.0', 'end'))
        self.fájlnév = fájlnév
```

Adjuk meg a tagfüggvényt a megfelelő menüpont `command` paramétereként, és teszteljük a programunkat!

A fájl megnyitása hasonlít a betöltéshez. Ugyanazt a párbeszédablakot használjuk.

Törlünk, majd betesszük a fájl tartalmát.

```
def fájl_megnyit(self):
    fájlnév = tkinter.filedialog.askopenfilename(
                defaultextension='.txt',
                filetypes=[('Szövegfájlok', '*.txt'),
                           ('Minden fájl', '*.*')])
    if fájlnév != '':
        with open(fájlnév) as forrásfájl:
            self.szövegterület.delete('1.0', 'end')
            self.szövegterület.insert('1.0', forrásfájl.read())
        self.fájlnév = fájlnév
```

A címsor

A programok címsorában szokás megadni az épp szerkesztett fájl nevét és az alkalmazás nevét. Mi is így teszünk, egy újabb tagfüggvénnyel. A tagfüggvény az `os` modulra támaszkodik a fájlnév és az elérési út szétválasztásakor, ne feledjük ezt a modult is importálni!

```
def ablakcímet_állít(self):
    if self.fájlnév:
        cím = os.path.basename(self.fájlnév)
    else:
        cím = 'névtelen'
    cím = cím + ' - Sajáttömb'
    self.master.title(cím)
```

A főablak címsorát állítjuk.

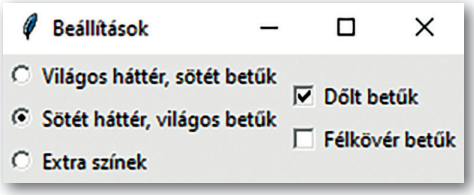
Helyezzük el az új tagfüggvény hívását mindhárom, fájlokkal foglalkozó metódusunk legvégére!

Ha minden jól ment, remekül működik az első grafikus felületű alkalmazásunk – kivéve a *Beállítások* menüt, amely külön leckét érdemel.

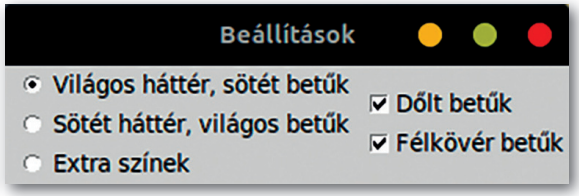
A Sajáttömb beállításai

Ebben a leckében a *Sajáttömb* egy új ablakot kap, amelyben a szövegterület színsémája, illetve a szövegterületen használt betűkészlet jellemzői állíthatók. A *Beállítások* ablakot az eddigieknek megfelelően, az `App` osztály tagfüggvényeként valósítjuk meg.

Két új widget



▶ A Beállítások ablak Windowson



▶ A Beállítások ablak Linuxon

Látjuk, hogy az ablakban rádiógombokkal oldottuk meg a színsémák kiválasztását. **Rádiógombokat** olyankor használunk, amikor a felkínált lehetőségek közül csak egy választható – ahogy a rádión is egyszerre egy csatornát tudunk csak hallgatni. (A mai rádióknak persze a legritkább esetben néznek ki így a gombjaik – ha egyáltalán vannak –, de amikor a rádiógombok megjelentek a GUI-kon, még könnyebben értelmezhető volt a név.)

A rádiógombok egy csoportot alkotnak. Úgy különülnek el a betűkészlet beállításaival foglalkozó jelölőnégyzetektől, hogy az ablakban egymás mellett két **keretet** (frame) helyeztünk el. A kereteket azért nem látjuk, mert létrehozásukkor nem intézkedtünk a hátterükként szolgáló ablaktól való elkülönülésükről. A bal oldali keretbe kerültek a rádiógombok, a jobb oldaliba a jelölőnégyzetek.

Jelölőnégyzeteket olyankor használunk, ha a választható lehetőségek külön-külön is bekapcsolhatók. A jelölőnégyzeteket egyre fokozódó ütemben cserélik az alkalmazások tervezői az érintőképernyővel jobban használható – és így a felhasználók számára a telefonjaikról ismerősebb – **csúszkákra.**

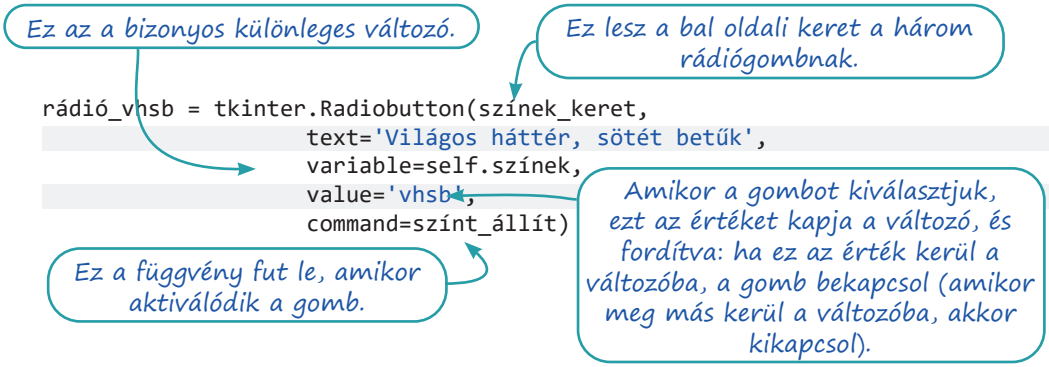
A rádiógombjaink

A három rádiógomb önálló widget, önálló `Radiobutton` osztályú objektum. Mégis valamiként értesülniük kell egymás állapotáról, hiszen ha az egyiket bekapcsoljuk, annak, amelyik eddig be volt kapcsolva, automatikusan ki kell kapcsolnia. Ráadásul a ki-be kapcsolásnak működnie kell akkor is, ha nem a felhasználó kapcsolta át a beállítást másik gombra, hanem a programunk „egyedül” tette ezt meg valamilyen okból, a felhasználó közreműködése nélkül. Bár most nem fogjuk megírni, de például elképzelhető, hogy induláskor betöltjük az utolsóként használt színsémát, és a megfelelő rádiógombot is aktiváljuk.

A gombok egy, a létrehozásukkor paraméterként megadott, közös változó segítségével tartják egymással a kapcsolatot. Létrehozásukkor megadjuk azt is, hogy a változó mely értéke esetén kapcsolódjanak be, minden más érték esetén kikapcsolva maradnak.

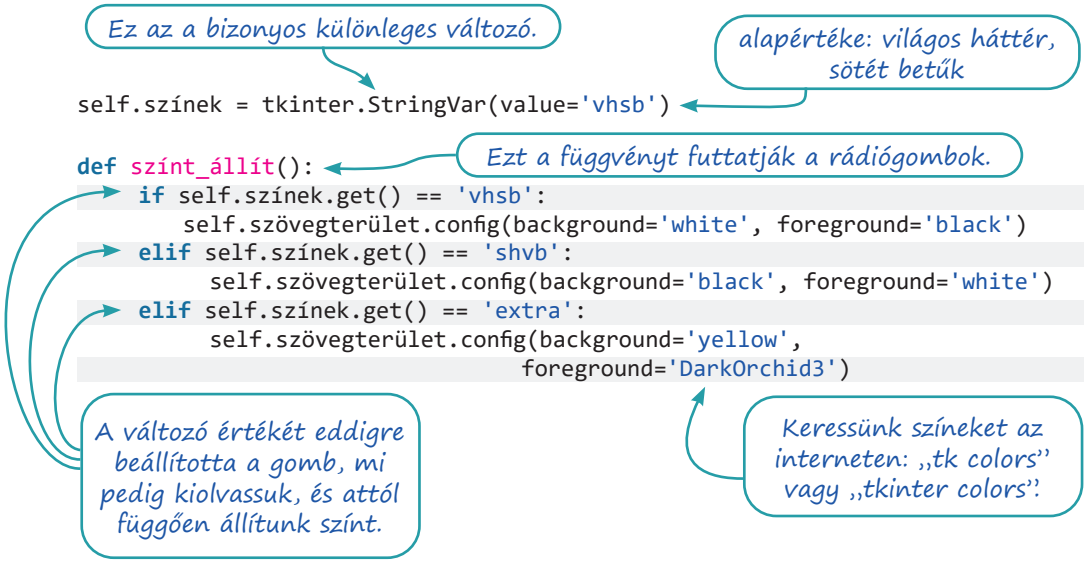
Ez a változó különleges változó, mert az értékének változásakor létrejövő **eseményről** a hozzá kapcsolódó gombok értesülnek. A gombok kódját pedig úgy írták meg, hogy az esemény bekövetkeztekor vizsgálják felül saját állapotukat, és szükség esetén változtassák meg.

Lássuk az első rádiógombunkat kialakító kódot:



Ilyen gombból helyezünk el hármat, csak a másik kettőnek más a neve, és más a kiírt szövege. Mindhármuknak ugyanaz a változója (a `színek`), de más-más értéket kapnak.

A lefuttatandó függvénynek nem tudunk paramétert átadni, és ahogy azt már az előző leckében előrevetítettük, ez most problémát jelent. Más-más színsémát kell beállítani a három gombnak, tehát vagy három külön tagfüggvényt írunk (ez a kód karbantarthatósága miatt nem jó ötlet), vagy helyettesítenünk kell a függvény paraméterezését. Íme a megoldás:



Látható, hogy sikerült olyan megoldást találnunk, amellyel nagyon könnyen vehetők fel új színek. Mindössze annyi a dolgunk, hogy elhelyezünk újabb rádiógombokat, és újabb `elif`-ágakkal bővítjük az `App.szint_állít()` tagfüggvényt.

A jelölőnégyzeteink

Míg a három rádiógombnak össze kellett dolgoznia, a jelölőnégyzeteink (`Checkbutton` osztályú objektumok) külön életet élnek: a betűk dőltsége nem függ össze a betűk vastagságával. Mégis szükség lesz az előzőhöz hasonló különleges változókra – jelölőnégyze-

tenként egyre. Azért kellenek, mert tárolnunk kell, hogy a jelölő épp be vagy ki van-e kapcsolva, illetve az érdemi munkát végző, a betűk tulajdonságát állító függvénynek most sem tudunk paramétert átadni, azaz ebből a változóból tudja meg, hogy ki- vagy bekapcsoltuk-e a jelölőt. Ezek a változók egész számot tárolnak, a 0 a kikapcsolt és az 1 a bekapcsolt állapotot jelenti.

A Beállítások ablak megvalósítása

A betűk tulajdonságainak állítása miatt szükség lesz a `tkinter.font` modulra, ezért importáljuk. Helyezzük el a három változónkat az `__init__()` tagfüggvény végén:

```
self.színek = tkinter.StringVar(value='vhsb')
self.dőlt = tkinter.IntVar(value=0)
self.félkövér = tkinter.IntVar(value=0)
```

Maga a tagfüggvény pedig a következő:

```
def ablak_beállítások(self):

    def színt_állít():
        if self.színek.get() == 'vhsb':
            self.szövegterület.config(background='white', foreground='black')
        elif self.színek.get() == 'shvb':
            self.szövegterület.config(background='black', foreground='white')
        elif self.színek.get() == 'extra':
            self.szövegterület.config(background='yellow',
                                      foreground='DarkOrchid3')

    def dőltet_kapcsol():
        betű = tkinter.font.Font(font=self.szövegterület.cget('font'))
        if self.dőlt.get() == 1:
            betű.configure(slant=tkinter.font.ITALIC)
        elif self.dőlt.get() == 0:
            betű.configure(slant=tkinter.font.ROMAN)
        self.szövegterület.config(font=betű)

    def félkövéret_kapcsol():
        betű = tkinter.font.Font(font=self.szövegterület.cget('font'))
        if self.félkövér.get() == 1:
            betű.configure(weight=tkinter.font.BOLD)
        elif self.félkövér.get() == 0:
            betű.configure(weight=tkinter.font.NORMAL)
        self.szövegterület.config(font=betű)
```

Ezt a függvényt már ismerjük.

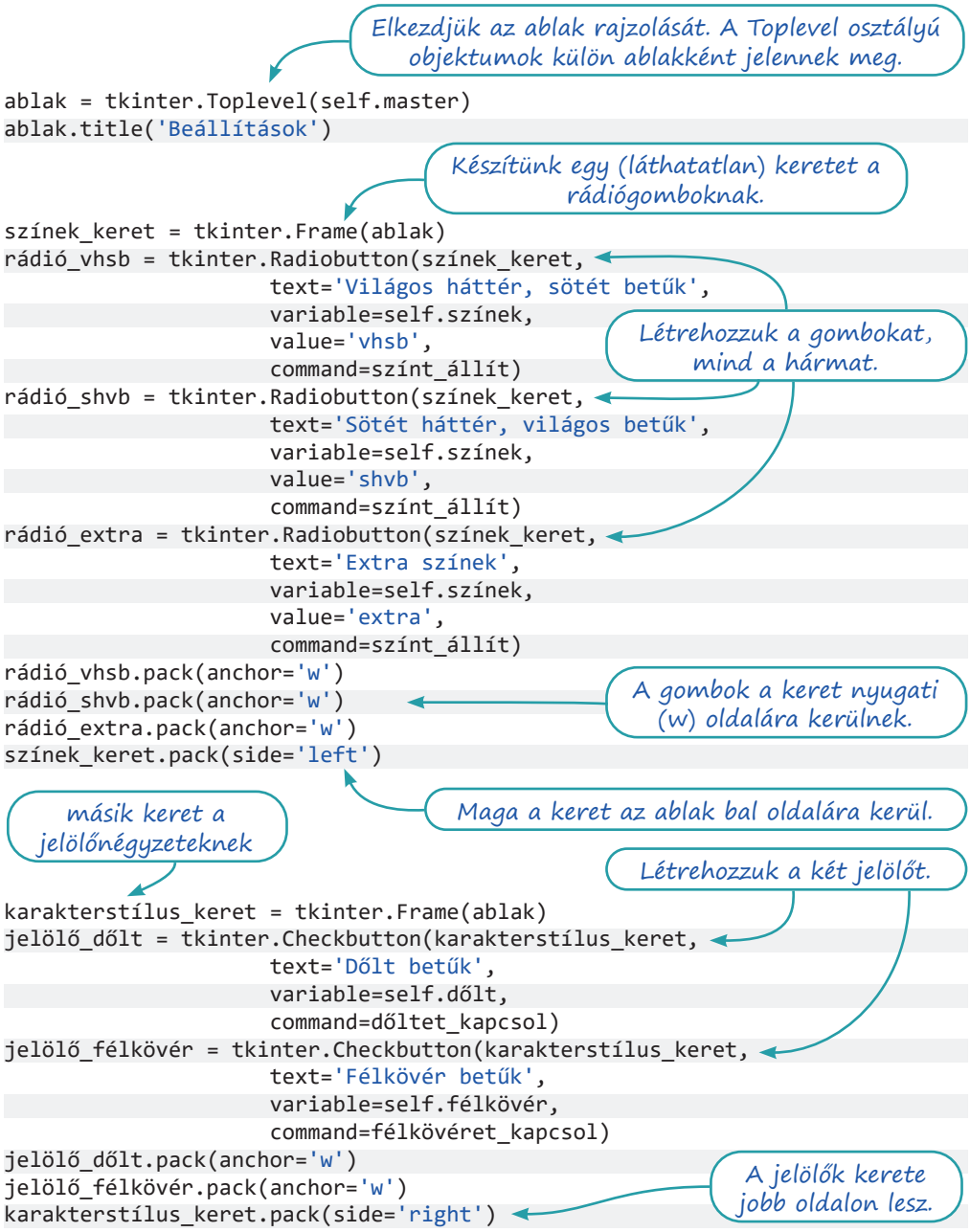
Ez a függvény kapcsolja ki-be a betűk dőltségét.

Kiolvassuk, hogy most milyen a betű (típus, méret).

A self.dőlt változó értékétől függően állítunk dőltet vagy egyeneset. A típus és a méret olyan lesz, amilyet kiolvastunk.

Ez a függvény állítja a betűk vastagságát.

A self.félkövér változó értékétől függően állítunk vastagabbat vagy vékonyabbat.

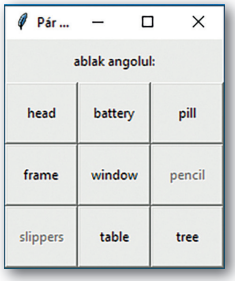



```
ablak = tkinter.Toplevel(self.master)
ablak.title('Beállítások')

színek_keret = tkinter.Frame(ablak)
rádió_vhsb = tkinter.Radiobutton(színek_keret,
                      text='Világos háttér, sötét betűk',
                      variable=self.színek,
                      value='vhsb',
                      command=színt_állít)
rádió_shvb = tkinter.Radiobutton(színek_keret,
                      text='Sötét háttér, világos betűk',
                      variable=self.színek,
                      value='shvb',
                      command=színt_állít)
rádió_extra = tkinter.Radiobutton(színek_keret,
                      text='Extra színek',
                      variable=self.színek,
                      value='extra',
                      command=színt_állít)
rádió_vhsb.pack(anchor='w')
rádió_shvb.pack(anchor='w')
rádió_extra.pack(anchor='w')
színek_keret.pack(side='left')

karakterstílus_keret = tkinter.Frame(ablak)
jelölő_dőlt = tkinter.Checkbutton(karakterstílus_keret,
                      text='Dőlt betűk',
                      variable=self.dőlt,
                      command=dőltet_kapcsol)
jelölő_félkövér = tkinter.Checkbutton(karakterstílus_keret,
                      text='Félkövér betűk',
                      variable=self.félkövér,
                      command=félkövéret_kapcsol)
jelölő_dőlt.pack(anchor='w')
jelölő_félkövér.pack(anchor='w')
karakterstílus_keret.pack(side='right')
```

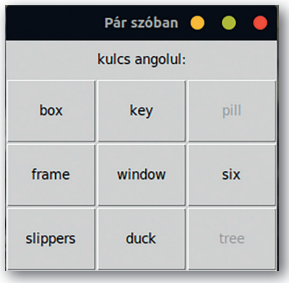
Utolsó feladatunk a tagfüggvény hívásának beállítása a megfelelő menüpont helyes paraméterezésével. Ha minden rendben működik, bátran kiörvendezhetjük magunkat az első „igazi” alkalmazásunk elkészülte alkalmából!

Grafikus felhasználói felületű alkalmazást fejlesztünk – A Pár szóban

A *Pár szóban* egy szótanuló alkalmazás. Induláskor beolvas egy szövegfájlt, benne soronként egy magyar szóval és az angol megfelelőjével. Kiválaszt közülük kilencet, és egyetlen szótárként (a magyar szavak a kulcsok) átadja őket az alábbi ablaknak:



► A Pár szóban Windowson



► A Pár szóban Linuxon

Az alkalmazás ablakának kialakítása

Az alkalmazás kiválaszt egyet a szótár elemei közül, amelyre vonatkozóan felteszi az ablak felső részén látható kérdést. A szótár értékei alapján létrehozza a kilenc gombot, melyeket soronként hármasával elhelyez az ablak alsó részén. Az egyes gombok neve megegyezik a rajtuk olvasható angol szóval.

A kérdezett szó megjelenítésére **címkét,** azaz `Label` osztályú objektumot használunk. Olyan pár szavas, pár mondatos szövegek megjelenítésére szolgál, amelyeket a felhasználó nem tud átírni.

Egy-egy widget elhelyezésére eddig a `tkinter` osztályainak `pack()` tagfüggvényét használtuk, amely elég egyszerűen működik ahhoz, hogy különösebb magyarázat nélkül is boldoguljunk vele. Ezúttal a bonyolultabb elrendezéseket lehetővé tévő `grid()` tagfüggvényt is bevetjük. Használatakor sorok és oszlopok jönnek létre az ablakon vagy kereten belül, és meg kell adnunk, hogy melyik oszlopba és melyik sorba kerüljön a widget. Grid, azaz **rács** használatával jelenítjük meg a `Button` osztályból példányosított gombokat.

Az eddig leírt működést megvalósíthatjuk az alábbi kóddal:

```
 1. import tkinter
 2. import random
 3.
 4. def fájlt_beolvas():
 5.     szójegyzék = {}
 6.     with open('szavak.txt') as forrásfájl:
 7.         for sor in forrásfájl:
 8.             magyar, angol = sor.strip().split()
 9.             szójegyzék[magyar] = angol
10.     megmaradó_szavak = random.sample(list(szójegyzék), 9)
11.     szójegyzék = { kulcs: érték for kulcs, érték in
12.                    szójegyzék.items() if kulcs in megmaradó_szavak }
13.     return szójegyzék
14.
```

Külön függvényként, valósítjuk meg, mert eléggé elkülönül a lényegi résztől.

Kiválasztunk kilenc szót, amelyeket megtartunk.

Szótárértelmezés. Megvalósítható hagyományos ciklusként is.

```
15. class App():
16.     def __init__(self, master, szójegyzék):
17.         self.szójegyzék = szójegyzék
18.         self.keresett_szó = random.choice(list(self.szójegyzék.keys()))
19.         self.kérdés_keret = tkinter.Frame(master)
20.         self.kérdés_keret.pack()
21.         self.kérdés = tkinter.Label(self.kérdés_keret,
22.                                     text=self.keresett_szó + ' angolul:',
23.                                     pady=10)
24.         self.kérdés.pack()
25.         self.gombok_keret = tkinter.Frame(master)
26.         self.gombok_keret.pack()
27.         self.gombokat_készít()
28.         self.gombokat_megjelenít()
29.
30.     def gombokat_készít(self):
31.         self.gombok = {}
32.         for angol in self.szójegyzék.values():
33.             self.gombok[angol] = tkinter.Button(self.gombok_keret,
34.                                                 text=angol,
35.                                                 height=3, width=8)
36.
37.     def gombokat_megjelenít(self):
38.         megjelenítendő_gombok = list(self.gombok)
39.         for sor in range(3):
40.             for oszlop in range(3):
41.                 aktuális_gomb = megjelenítendő_gombok.pop()
42.                 self.gombok[aktuális_gomb].grid(row=sor, column=oszlop)
43.
44. főablak = tkinter.Tk()
45. főablak.title('Pár szóban')
46. app = App(főablak, fájlt_beolvas())
47. főablak.mainloop()
```

Az első keresett szó.

Két keretünk lesz egymás fölött. A felsőbe kerül a kérdés, az alsóba a gombok.

A kérdés rákerül a címkére.

Előbb elkészítjük mind a kilenc gombot…

…aztán kitesszük őket.

A gombok egy szótárba kerülnek.

Az angol szó kerül rájuk.

Ekkorák lesznek.

Itt készül a 3×3-as formáció.

A gombot a pop() kiveszi a még megjelenítendő gombok közül, és értékül adja az aktuális_gomb változónak.

Itt adjuk át a lényegi résznek a kiválasztott szópárokat.

Az alkalmazás működni kezd

Két tagfüggvénnyel bővítjük az alkalmazás `App` osztályát. Az első a gombok lenyomásakor összehasonlítja a gomb nevét (azaz a rajta lévő angol szót) a felső részen olvasható szó angol megfelelőjével. Ha azonosak, akkor kikapcsoljuk a gombot, hogy ne lehessen újra lenyomni. A másikkal végignézzük a szójegyzékünket, illetve azokat a gombokat, amelyek még lenyomhatók, és összevetve őket új szót írunk ki felülre.

Persze a két tagfüggvénynek csak akkor van értelme, ha hívjuk is őket. Az elsőt, az egyezést vizsgálót kell hívnunk a megfelelő gomb lenyomásakor, és ha jó gombot nyomott a felhasználó, akkor majd ez a függvény hívja a másikat, amelyik kiírja az új kérdést.

A függvényhívást a gombok `command` paraméteréül kell megadnunk a 35. sor zárójelén belül. Természetesen most sem tudunk paramétert átadni a függvénynek, pedig jó volna az egyezést vizsgáló függvény tudomására hozni, hogy melyik gombot nyomta le a felhasználó. Az előző leckében megismert különleges változóink helyett ezúttal a korábban futólag említett lambda-függvényen alapuló trükkel leszünk úrrá a problémán. A paraméter:

```
command=lambda angol_szó=angol: self.egyezést_vizsgál(angol_szó)
```

A két tagfüggvény:

Ezt a szót kaptuk paraméterül a lenyomott gombtól.

Ha megegyezik a felülre kiírt szóval...

```
44.     def egyezést_vizsgál(self, szó):
45.         if szó == self.szójegyzék[self.keresett_szó]:
46.                 self.gombok[szó].config(state='disabled')
47.                 self.keresett_szót_frissít()
48.
```

...akkor kikapcsoljuk a gombot, és hívjuk az új kérdést feltevő függvényt.

A még ki nem kapcsolt gombokon lévő szavakat kérdezhetjük.

```
49.     def keresett_szót_frissít(self):
50.         még_nem_kérdezett_szavak = []
51.         for magyar, angol in self.szójegyzék.items():
52.             if self.gombok[angol]['state'] != 'disabled':
53.                 még_nem_kérdezett_szavak.append(magyar)
```

Ha találtunk még kérdezhető szót...

...akkor kérdezünk...

```
54.         if még_nem_kérdezett_szavak:
55.             self.keresett_szó = random.choice(még_nem_kérdezett_szavak)
56.             self.kérdés.configure(text=self.keresett_szó + ' angolul:')
57.         else:
58.             self.kérdés.configure(text='Kifogytam a szavakból.')
```

*...különben **elbúcsúzunk**.*

Elérve a tankönyvsorozat programozásról szóló leckéi közül az utolsó végére, remélhetőleg senki számára sem teljes boszorkányság többé a számítógép programjainak működése. Sokkal pontosabb elképzeléseink lettek a bennünket mindennapjainkban körülvevő eszközök belső működéséről. Alighanem mindenki előtt, aki eddig velünk tartott, kitágult a horizont. Lesznek olyanok, akik már ezt is túlzásnak vélik, másokat pedig újabb, hosszabb felfedezőutakra csábít az, amit eddig megismertek csodálatos világunknak ebből a részéből.

Szerencsés utat!